Study and benchmarking of Artificial Intelligence (AI) model serving systems on edge computation units and cloud environments

Bote Liu

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 27.5.2019

Supervisor

Prof. Antti Ylä-Jääski

Advisor

Roberto Morabito

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Bote Liu

Title Study and benchmarking of Artificial Intelligence (AI) model serving systems on edge computation units and cloud environments

Degree programme Computer, Communication and Information Sciences

Major Mobile Computing, Services and Security **Code of major** SCI3045

Supervisor Prof. Antti Ylä-Jääski

Advisor Roberto Morabito

Date 27.5.2019 **Number of pages** 58+3 **Language** English

Abstract

Recently, the development of Edge AI, which brings computation resources and artificial intelligence closer to the network edge, has been growing rapidly. Compared to Cloud AI applications, Edge AI takes advantage of low network latency and reduced bandwidth consumption. However, most of the Edge AI applications are still in the conceptual phase, and thorough implementations are not well achieved. In addition, the performance impact of the AI model serving that moves from cloud to edge devices – which can be resource-constrained – has not been sufficiently investigated. This thesis aims to fill the gap in this respect, by investigating and benchmarking edge-based and cloud-based AI model serving, through the use of existing serving systems such as TensorFlow Serving and Clipper. Furthermore, we rely on the usage of Kubernetes for ensuring more robustness and higher scalability both on edge and cloud – we use a Single-Board Computer as edge device, and Google Kubernetes Engine (GKE) as cloud environment. From the empirical point of view, we characterize the performance of Edge AI and Cloud AI model serving in terms of throughput and response time, by taking into account several AI applications that are based on classical machine learning and deep learning algorithms. Our results show that, as expected, Edge AI benefits from lower network latency, while Cloud AI benefits from higher computational resources availability. However, we found out that the an edge-based approach introduces some advantages when serving high-throughput applications, when caching mechanisms are used, and when a few users (preferably one) query the model serving.

Keywords AI model serving, edge AI, cloud computing, kubernetes, microservices, Tensorflow Serving, Clipper, GKE

Preface

I would like to express my sincere gratitude to Ericsson LMF, which provides me with such a good chance to explore a new field that I am taking a keen interest in. In addition, I am indebted to my thesis advisor, Roberto Morabito, for sparing time to revise my thesis carefully and give me consistent encouragement throughout the thesis work. I am also grateful to Carlos R. B. Azevedo for his knowledge and advice on AI/ML algorithms and models. My thanks are extended to Edgar Ramos and Matvej Yli-Olli for their help and suggestions on my thesis writing. Finally, I want to thank my thesis supervisor, Professor Antti Ylä-Jääski and Ericsson line managers, Pasi Holkko and Peter von Wrycza for their generous support for my thesis work.

Espoo, 27.5.2019 Liu, B.

Contents

Abstract 3

Preface 4

Contents 5

Abbreviations 8

1 Introduction 9
- 1.1 Cloud AI and edge AI . . . 10
- 1.2 Research questions . . . 12
- 1.3 Thesis outline . . . 12

2 Background 13
- 2.1 Cloud computing . . . 13
- 2.2 Edge computing . . . 13
- 2.3 Machine learning . . . 14
- 2.4 AI frameworks . . . 14
  - 2.4.1 Tensorflow . . . 15
  - 2.4.2 Scikit-learn . . . 15
- 2.5 AI models . . . 15
  - 2.5.1 SSD inception v2 . . . 15
  - 2.5.2 Logistic regression and random forest . . . 16
  - 2.5.3 The selection of AI models . . . 16
- 2.6 AI model serving . . . 16
  - 2.6.1 AI model serving on cloud . . . 16
  - 2.6.2 AI model serving on edge . . . 17
- 2.7 AI model serving systems . . . 17
  - 2.7.1 Tensorflow Serving . . . 17
  - 2.7.2 Clipper . . . 18
- 2.8 Container virtualization and orchestration for microservices . . . 18
  - 2.8.1 Docker container . . . 18
  - 2.8.2 Kubernetes . . . 19
- 2.9 Summary . . . 20

3 Methodology 21
- 3.1 Docker . . . 21
- 3.2 Acquire AI trained models . . . 21
  - 3.2.1 SSD inception v2 (COCO) . . . 22
  - 3.2.2 Logistic regression and random forest . . . 22
- 3.3 Setup of AI serving systems . . . 24
  - 3.3.1 Tensorflow Serving setup . . . 24
  - 3.3.2 Clipper setup . . . 27

3.4 AI model serving on GKE . . . 31
3.4.1 GKE setup . . . 31
3.4.2 Deploy AI serving with TensorFlow Serving on GKE . . . 32
3.4.3 Deploy AI serving with Clipper on GKE . . . 35
3.5 AI serving on UP board . . . 36
3.5.1 UP board setup . . . 37
3.5.2 Deploy AI serving with TensorFlow Serving on edge . . . 37
3.5.3 Deploy AI serving with Clipper on edge . . . 38
3.6 Performance measurement method . . . 38
3.6.1 Prepare request payload . . . 38
3.6.2 Measure performance of serving . . . 40
3.7 Summary . . . 41

4 Results and Analysis **42**
4.1 Model SSD-inception-v2 (COCO) . . . 43
4.2 Model logistic regression . . . 47
4.3 Model random forest . . . 48

5 Conclusion and discussion **51**

References **54**

A Python script example to measure the performance of AI model serving **60**

List of Figures

1 The relation of training, inference (prediction) and model serving . . . 9
2 Edge computing brings computation resources close to where data is produced. . . . 11
3 AI serving deployed with Tensorflow Serving on GKE cluster . . . 33
4 AI serving deployed with Clipper on GKE cluster . . . 35
5 AI serving deployed with Tensorflow Serving on edge . . . 38
6 AI serving deployed with Clipper on edge . . . 39
7 The distribution of the COCO image resolutions . . . 40
8 The method to measure the performance of AI model serving . . . 41
9 Throughput of the AI model serving with the model “SSD inception v2 (COCO)” and request payload from CIFAR-10 . . . 43
10 Response time of the AI model serving with the model “SSD inception v2 (COCO)” and request payload from CIFAR-10 . . . 45
11 Throughput of AI model serving with the model “SSD inception v2 (COCO)” and request payload from COCO . . . 46
12 Response time of AI model serving with the model “SSD inception v2 (COCO)” and request payload from COCO . . . 46

13 Throughput of AI model serving with the model “logistic regression” and request payload from MNIST 48
14 Response time of the AI model serving with the model “logistic regression” and request payload from MNIST 48
15 Throughput of the AI model serving with the model “random forest” and request payload from MNIST 49
16 Response time of the AI model serving with the model “random forest” and request payload from MNIST 49
17 Throughput of the AI model serving with the model “random forest” and request payload from MNIST (one user, one pod) 50

List of Tables

1 The resource configurations of two deployment environments 42
2 The basic overhead to initialize the environments 43
3 CPU utilization of the two nodes in GKE cluster with model “SSD inception v2 (COCO)” and request payload from CIFAR-10 44
4 Memory utilization of the two nodes in GKE cluster with model “SSD inception v2 (COCO)” and request payload from CIFAR-10 44
5 CPU utilization of UP board with model “SSD inception v2 (COCO)” and request payload from CIFAR-10 44
6 Memory utilization of UP board with model “SSD inception v2 (COCO)” and request payload from CIFAR-10 44
7 CPU utilization of the two nodes in GKE cluster with model “SSD inception v2 (COCO)” and request payload from COCO 47
8 Memory utilization of the two nodes in GKE cluster with model “SSD inception v2 (COCO)” and request payload from COCO 47
9 CPU utilization of UP board with model “SSD inception v2 (COCO)” and request payload from COCO 47
10 Memory utilization of UP board with model “SSD inception v2 (COCO)” and request payload from COCO 47

Abbreviations

OS	Operating System
LAN	Local Area Network
GCP	Google Cloud Platform
GKE	Google Kubernetes Engine
CPU	Central Processing Unit
GPU	Graphics Processing Unit
TPU	Tensor Processing Unit
AI	Artificial Intelligence
k8s	Kubernetes
SSD	Single Shot Detector
COCO	Common Objects in Context (a large-scale object detection, segmentation, and captioning dataset)
MNIST	Mixed National Institute of Standards and Technology (a database of handwritten digits used for training image processing systems)
CIFAR	Canadian Institute For Advanced Research (a collection of images that are commonly used to train machine learning and computer vision algorithms)
MEC	Mobile Edge Computing
CNN	Convolutional Neural Network
L4	Layer 4
L7	Layer 7
AWS	Amazon Web Service
AKS	Azure Kubernetes Service
NIST	National Institute of Standards and Technology
YAML	YAML Ain't Markup Language
REST	Representational State Transfer
API	Application programming interface
AVX	Advanced Vector Extensions
DNS	Domain Name System
gRPC	Google Remote Procedure Call

1 Introduction

In recent years, the applications driven by machine learning have captured more and more attention. Machine learning can help machines on learning from the past experience and improve the decisions that they will make in the future. Figure 1 shows the main processes and components characterizing the machine learning. The process by which machines are learning is called **training** or **learning**, while the phase in which machines take decisions or make predictions after training is called **Inference**. The machine learning **model** is the output of the training process and it is used to make predictions. **Model Serving** is the process of taking a trained model and making it available to serve prediction requests. Training is usually considered as a complex process in which a huge amount of computational resources occur with a high amount of data needed, and it may even take from a few hours to several days for completing a training process [29][30]. Differently, inference is often considered a more lightweight computational work, even though inference requests can be requested from around the world.

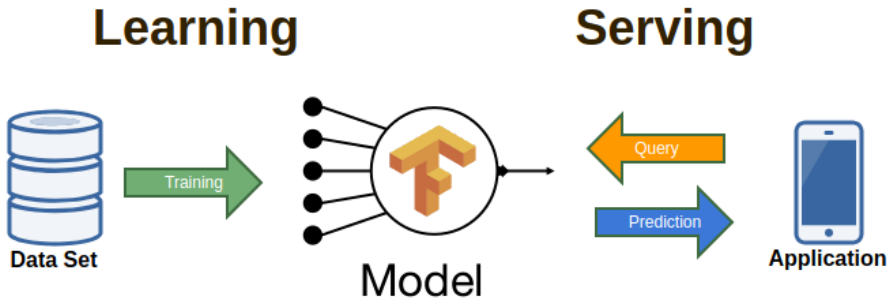

Figure 1: The relation of training, inference (prediction) and model serving

As an example, we can consider the Amazon's products recommendation functionality. These kind of services are supposed to rapidly scale because of the high number of users connected, but also to handle high peaks of requests (e.g. on Black Friday day). Moreover, the product recommendations should be given timely and accurately as the user interests change.

The challenges of training and inference are different. For the training part, researchers and developers usually select a model and the use of a machine learning framework that is suitable to the specific model and the preferred programming language, by taking into account also hardware availability and suitability. Then the training is started with the tools provided by the selected machine learning framework. It is very rare that the trained model can provide good performance at the first attempt so further iterations with a different set up of the parameters are needed. Comparing with model training which demands complicated infrastructure and theory, model serving (inference) puts more focus on practice and user experience. Deployers need to prepare the runtime of the trained model according to the machine

learning framework by which the model was trained. In addition, the model serving is expected to scale rapidly as the growing number of users. High throughput, low latency and high availability are necessities of a good model serving. Furthermore, higher rate of deployment automation and less maintenance are also strongly desirable.

Responding to the challenges of machine learning training, many new frameworks are being developed focusing on specific models or specific application domains. For example, Tensorflow [22] is well-known for handling deep learning models, Vowpal Wabbit [25] for large linear models, Caffe [23] for image classification, HTK [24] for speech recognition, etc. An increasing number of new machine learning algorithms, models and tools – such as [26][27] – are being designed, developed, and updated constantly. However, although there is a relevant effort on developing further and optimizing the training phase, there has not been so far the same attention for enhancing the inference phase (model serving). As the majority of the AI applications (e.g. facial payment, automated driving system, etc.) require high precision and reliability, there is a growing demand for an effective, highly robust, scalable and responsive model serving mechanisms. In order to cope with the higher demand of the inference phase, this thesis aims to characterize the AI model serving, by considering several scenarios and different use cases.

1.1 Cloud AI and edge AI

In recent years, the development of AI applications has been growing rapidly. From online shopping service, to peer-to-peer ridesharing platform, AI has entered into almost every aspect of our lives. Many AI services are deployed on cloud computing facilities because AI or machine learning algorithms usually entail a large amount of computation and resource consumption demand, and cloud services can meet the requirement with the allocation of flexible resources which can be assigned on demand. However, AI applications are characterized by heterogeneous configuration and performance requirements. This heterogeneity implies also that, in some cases, cloud-based solution cannot always satisfy at once the requirements of all the AI applications, especially from performance perspectives. As an example, an AI cloud-based automated driving system could face several issues on satisfying the very strict latency requirements that these types of service are demanding. In fact, the time spent on the decision making of the driving system is heavily dependent on the network situation and cloud-based approach cannot be considered the most effective approach for avoiding adverse network conditions. The implications of late responsive systems can undermine the safety of the passengers, when for example the car is entering into a remote area characterized by limited connectivity or the network is temporarily congested. In addition, relying on third parties cloud services can lead to privacy issues, when personal data and street-view images that may include the faces of the pedestrians nearby are sent and stored in the cloud. Besides, Internet access is not always available. Considering all these problems and limitations, the concept of **Edge Computing** has recently emerged. The new approach aims to extend computation resources closer to where data is produced, as Figure 2 shows.

Also AI can substantially benefits from an edge-computing based approach. One advantage lies in the fact that the physical distance between end-devices and edge is considerably less when compared to the distance between a data center – which can provide services from remote regions compared to where the requests come from – and the end-device that is receiving the service. A clear advantage deriving from the reduced physical distance is a reduced network latency, which is critical for many AI applications such as automated driving system. Besides, the privacy issue can be mitigated because AI requests are processed locally, and users might not need to trust on third party services.

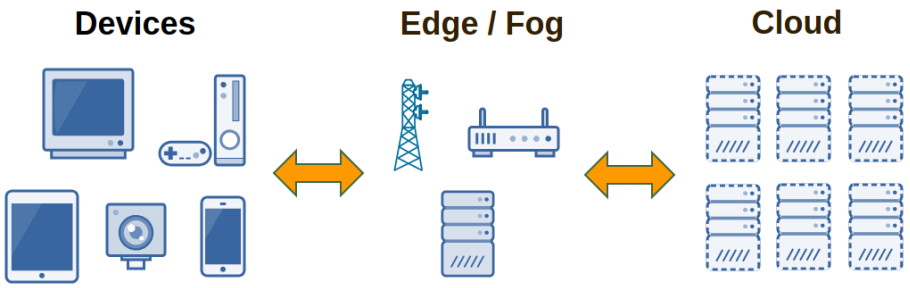

Figure 2: Edge computing brings computation resources close to where data is produced.

Considering the numerous advantages of edge computing [5][6], which can enable many applications and services [7][8][9], a great number of research work has been done in this area. One interesting research area is Mobile Edge Computing (MEC) [1][2]. The rising study of MEC has been undertaken in order to achieve the goal of offloading the backbone network workload, by moving the cloud computing resources and capabilities to the edge of the cellular network [11][10]. Instead of moving computation resources to the cellular network, Ren et al [4] proposed a new vision of edge computing, particularly suitable for IoT networks, using transparent computing. The idea of IoT-based edge computing is also advocated by Zhang et al, who deploy Big Data services on IoT networks in order to overcome the challenges and the limitations of cloud computing deployments[3]. Naturally, the merits of edge computing also capture a lot of attention of AI experts. Due to the problems and challenges of the traditional AI serving, the ideas of AI decentralization were brought up and well discussed in [12] and [13]. Furthermore, the concepts of edge computing and AI are combined to solve several problems in practice. [15] brings deep learning into edge environment implemented with IoT devices, and a resource-efficient edge computing scheme was proposed in [16], where compute-intensive tasks can be offloaded across the local devices and the edge cloud. [14] introduced a new fog computing architecture that facilitates infrastructure and services deployment in smart cities. Moreover, there has been a considerable amount of work focusing on model serving of a specific application, e.g. video recommendation [17], speech recognition [21], product recommendation [20] and targeted advertising [18][19], etc.

1.2 Research questions

The thesis activity aims to investigate and benchmark the deployment of different AI model serving options, by qualitatively and quantitatively evaluating existing systems such as TensorFlow Serving [50] and Clipper [28]. Additionally, one of the core part of the work is to make possible the execution of such serving systems on edge devices such as Single-Board Computer.

In summary, the work sought to answer the following research questions:

- How to deploy and expose AI model serving in Single-Board Computers?
- How this service exposure and deployment option compares with the cloud-centric view in terms of performance?

1.3 Thesis outline

Chapter 1 provides a high-level description of the investigated research area, highlighting also what are the key challenges characterizing it. Chapter 2 introduces the background information needed for getting familiar with the key concepts of the thesis. In chapter 3, the experimental setup is introduced in detail, while in Chapter 4 the results of our empirical evaluation are presented and analyzed. Finally, Chapter 5 provides the conclusion and the main insights deriving from this thesis, discussing also the possible future work.

2 Background

This chapter explains some concepts that are supposed to be known to follow up the content of the following chapters. After finishing this chapter, readers will understand why these concepts and topics are important and relevant in the thesis.

2.1 Cloud computing

As the definition from AWS, "Cloud computing is the on-demand delivery of compute power, database, storage, applications, and other IT resources via the internet with pay-as-you-go pricing" [31]. Cloud computing can also be regarded as a pool of computer resources that users can use on demand conveniently without direct maintenance and management themselves. Physical hardware resources like CPUs, memory, storage, network capacity, etc are virtualized on cloud for fine-grained virtual resources provisioning. If cloud is provided over the Internet and for unselected customers, it is **public cloud**. Specifically, Redhat defines public cloud [32] as "a pool of virtual resources—developed from hardware owned and managed by a third-party company—that is automatically provisioned and allocated among multiple clients through a self-service interface." Companies can benefit from public cloud because they save up the up-front infrastructure costs substantially and only the amount of virtual resources they actually used is charged. Startups like public cloud because they pay little for hardware in the beginning phase, however, they can rapidly adjust the resource configurations as the number of users is growing. In addition, public cloud is also suitable for users who want to make a test or experiment quickly. AWS, Azure and GCP are main players in the field of public cloud. In contrast, private cloud is usually for one organization and internal use. The benefits of private cloud are security and flexibility but the cost may be expensive. Hybrid cloud is a combination of public cloud and private cloud. For example, a mobile network operator has its own data center and it can rent out its unused resources to external users. In addition, cloud computing can also be categorized by service models as IaaS, PaaS and SaaS by NIST [33]. Briefly speaking, IaaS only manages and provides the underlying cloud infrastructure, PaaS provides application developers with development environments, and customers can use the provider's service from SaaS.

In our experiment, GKE [34] is chosen for the cloud environment. On the one hand, GKE is public cloud on which tests and experiments are easily made without management and maintenance of computer resources. On the other hand, GKE can start up a kubernetes cluster conveniently and save us a lot of time.

2.2 Edge computing

Edge computing is defined as "a networking philosophy focused on bringing computing as close to the source of data as possible in order to reduce latency and bandwidth use" [35] by CloudFlare. Edge computing moves the processing

of requests from servers on the cloud to edge devices such as IoT devices. Bring computing to edge or local reduces the network communication distance which is very important for some specific applications.

Take an automated driving system as an example. Assume that a camera in a self-driving car takes street-view pictures every second and sends them to a server on cloud for road conditions analysis. The server analyzes the pictures and sends back its analysis results such as driving directions. In this case the first problem is that the car could receive delayed results from the server due to network latency, and it may cause a serious accident. In addition, assume that in the future all the cars are equipped with the automated driving system and they send pictures to the server, the network bandwidth will be significantly consumed and the server is overwhelmed by these requests. Now imagine that the processing of picture analysis is moved to edge devices. Then the situation is much improved because the analysis results of pictures will be timely received due to short and stable network latency. Moreover, the bandwidth consumption of the Internet infrastructure is much less than before as most of the traffic is not outside the edge. To sum up, the benefits of edge computing are low network latency and low bandwidth usage, which are critical for some real-time applications.

2.3 Machine learning

Machine learning can help machines on learning from the past experience and improve the decisions that they will make in the future. The process that machines are learning is called **Training**, and the phase that machines make decisions or predictions after training is called **Inference**. People may wonder why machine learning is necessary, considering the existing algorithms and programs can already do many things for us now? There are three main reasons [36]. One reason that algorithms cannot replace machine learning is that the designers of the algorithms cannot anticipate all situations in a specific environment. For example, so far there is no such a traditional algorithm that can enable a robot vacuum to clean a messy house efficiently without making observations in this new environment. In addition, the designers of algorithms cannot anticipate all changes over time. It is difficult for a developer to write a piece of code that can predict the weather of the next day accurately without adapting its prediction with the latest meteorological data. Moreover, there are some functionalities that are very difficult to be implemented by a program, e.g. recognizing handwriting digits is challenging for a traditional algorithm, but machine learning can achieve a very high accuracy for this purpose.

2.4 AI frameworks

An AI framework is an interface, library or tool that makes it easy to build and train models. Thanks to AI frameworks, developers without the expert-level knowledge of underlying algorithms and details can construct and optimize models easily and conveniently. Tensorflow [22] and scikit-learn [37] as two common AI frameworks are

used to train AI models in the thesis. Tensorflow is used because it is the base of Tensorflow Object Detection API [39][38] which is of our interest, and scikit-learn is selected due to its convenience to build and train standard models (logistic regression and random forest, in our case).

2.4.1 Tensorflow

TensorFlow [22] is an end-to-end open source platform for machine learning [40]. It provides a library for numerical computation using dataflow graph, and is commonly used for neural network and deep learning. The library that Tensorflow provides is low-level and so it allows researches to design and test new models with a set of simple operators. With Tensorflow, models can be designed in a more flexible way, and GPUs are easily used for fast training.

2.4.2 Scikit-learn

Scikit-learn [37] is higher level library with which one can quickly build and train standard machine learning models with a few lines of python code. Therefore it is quite useful if you want to build and train a standard model quickly. However, scikit-learn is not very convenient to design and implement custom machine learning models.

2.5 AI models

An AI/ML model is the output of machine learning training. The model of "SSD inception v2" [45] is based on Convolutional Neural Network (CNN) [41], which is a class of deep neural networks that are mostly used for computer vision. This section gives a brief introduction to the three models that will be used in the following experiments. At the end of the section, the reasons to choose these models are also explained.

2.5.1 SSD inception v2

The model of "SSD inception v2" is a AI/ML model that is trained with the open source framework of **Tensorflow Object Detection API** [38], which is built on top of Tensorflow. This model is used to serve prediction requests for localizing and identifying multiple objects from one image. Besides "SSD inception v2", other CNN-based models like Fast R-CNN [42], R-FCN [43], Multibox [44] and YOLO [46] are also modern object detectors.

The inputs of the model are the pixel values of color images, and the inference of the model includes the following 4 results:

- Detection boxes, which localize the objects in the image.
- Detection classes, which classify the objects in the image.

- Detection scores that indicates confidence that the object was genuinely detected.
- The number of objects detected.

2.5.2 Logistic regression and random forest

Logistic regression [47] is a easy, fast and simple machine learning algorithm, and is a good choice to start with classification algorithms. The output of logistic regression is a probability (between 0 and 1), which can be used to predict binary results 0 or 1. If the probability is larger than 0.5, its prediction result is 1, otherwise it predicts 0. The binary logistic regression can be extended for multi-class classification as well.

Comparing with logistic regression, random forest [48][49] is a more accurate and powerful machine learning model. Random forest is an ensemble model where multiple decision trees are combined to form a more robust model. The model can give a more accurate result and tolerate relatively lower signal to noise ratio even if a single tree in the forest is highly sensitive to the noise in the training data set.

2.5.3 The selection of AI models

The three models introduced above are selected in the following experiments. The idea behind the selection is based on two factors. The first factor is that we want to choose models used in different scenarios. SSD inception v2 is based on CNN, which has remarkable performance on image classification, while logistic regression and random forest are classical machine learning models, and they are capable for classifications for general purposes. In addition, we want to use models with distinct characteristics, and in our case the time required for prediction is considered. SSD inception v2 is very slow as it is based on deep neural network, and logistic regression needs the least time. Random forest, which is relatively complex than logistic regression, ranks in the middle position.

2.6 AI model serving

AI model serving is a process of taking a trained model and making it available to serve prediction requests. The following sections give a brief introduction to AI model serving that is provided on cloud and edge respectively.

2.6.1 AI model serving on cloud

Cloud solutions for AI model serving are appealing. Cloud has rich resources for computation, storage, network capacity, etc, and AI model serving can use the cloud feature of autoscaling to adjust resource usage based on the varying traffic and workload. In addition, some hardware, e.g. GPU and TPU can be conveniently used for deploying AI models that are optimized for these hardware. Moreover, there are many AI related functions and services from cloud vendors, and the work of both AI

model training and serving is facilitated on the cloud platforms. However, cloud AI is not perfect and it faces challenges. One challenge is the significant consumption of Internet bandwidth due to the long distances that AI requests have to pass through. For the same reason, it is difficult for cloud AI to meet the requirements of some real-time AI applications for which timely responses are critical.

2.6.2 AI model serving on edge

AI model serving on edge, or **Edge AI** is one solution that is expected to overcome the challenges that cloud AI is facing. Edge AI pushes the complexity of AI processing away to the edge of the network. The benefits of Edge AI are many. First, it serves AI requests on the edge of the network, and this action offloads traffic from the backbone network to the local network. Another benefit of edge AI is that it decreases network communication distance and network latency, which are critical for some AI real-time applications, e.g. automated driving system. Furthermore, a potential problem with privacy, which exists in cloud AI, is mitigated as the traffic is controlled in a smaller area.

2.7 AI model serving systems

An AI model serving system provides model serving based on a given model. A serving system prepares the runtime of the model and provides serving for prediction requests with a trained AI/ML model. In the following part, two common model serving systems, Tensorflow Serving [50] and Clipper [28][51], are introduced.

2.7.1 Tensorflow Serving

Tensorflow Serving is a flexible, high-performance serving system for AI/ML models and it is designed for production environment [50]. It has been used in production for relative long time and its quality and stability are well tested and verified. There are quite good documentations for learners and developers, and it is also widely used and integrated in other machine learning tools such as Kubeflow [52]. It provides out-of-the-box integration with Tensorflow models, but needs more work to serve with other types of models.

Tensorflow Serving can run in a docker container or on a k8s cluster. In order to run Tensorflow Serving in a docker container, one needs to run a container with a docker image **tensorflow/serving** as the first step. Then copy a trained or downloaded model and variables into the container and update environment variables. The last step is to commit the container and a new docker image with the given model is created. If one wants to use Tensorflow Serving with k8s, a YAML [53] configuration file, orchestrating docker containers, is needed for this purpose.

2.7.2 Clipper

Clipper is another prediction serving system for machine learning [51]. Clipper can support different types of models conveniently, and it provides a possibility to add custom logic before prediction requests are processed by model containers.

Clipper also provides model serving in docker container or k8s cluster. To use Clipper, a typical way is to write a python script and configure the deployment with the following steps:

- Configure Clipper with docker/k8s manager and start Clipper.
- Define a AI/ML model by defining a python closure.
- Register an application, creating and exposing a prediction REST API.
- Configure and deploy the defined model in docker containers.
- Link the registered application and the deployed model.

2.8 Container virtualization and orchestration for microservices

Microservices [57][58] is a software development method that decouples an application into several loosely services [56]. It provides a benefit of modularity, which makes the work of development, testing and deployment conveniently. In addition, unlike the monolithic approach in which an application supporting multiple functions has to scaled in a way that all functions are scaled as a whole even if only one function needs to be scaled, microservices can easily scale the services of an application separately. It is very common to apply microservices to cloud-native applications [56], which are based on container technology. In the container technology, docker and kubernetes are very popular and widely used in many fields.

2.8.1 Docker container

Docker [54] is one type of container, which isolates from each other and has its own software, libraries and configurations individually. Containers share one OS kernel and therefore are more lightweight than virtual machines [55], each of which has its own OS. The function and behavior of a docker container is defined by a docker image, which can be stored and downloaded with a public container image repository, e.g. docker hub [59], or a private docker registry.

Container has a lot of merits, e.g. isolation, cross-platform and lightweight, which enable it to be used in a wide range. In addition, a container provides an environment in which all dependencies are ready for the application inside the image, and it can run in wherever docker engine is installed. These advantages abstract away the complexity of the running environments, and largely facilitates the deployment of products, which makes it very popular in software industry.

2.8.2 Kubernetes

Kubernetes or k8s [60], is a container orchestration tool and provides a container-centric management environment. K8s facilitates the automation of application deployment across a cluster of hosts or virtual machines. In addition, with k8s, it is easy to manage deployment versions and scale the services of applications.

Kubernetes cluster setup

For single node, minikube [61] and microk8s [62] are good choices to try out the k8s cluster. Minikube is suitable for running a k8s cluster in a VM on a laptop, while microk8s can start up a k8s cluster under 60 seconds on about any linux box. In the following experiments, microk8s is chosen to setup a k8s cluster on a single-board computer because it has an advantage over minikube that it doesn't require a hypervisor.

In addition, cloud vendors provide their own managed services for k8s, which enable users to fast deployment and easy maintenance. For example, on top of GCP [63], Google launched GKE [34] to fast deploy k8s cluster. Besides, Amazon ECR and Azure AKS are the counterparts on other cloud platforms.

There are also custom solutions on cloud platforms or on-premises, but they will not be introduced here because they are not very relevant to the thesis content.

Pod

A pod is a basic unit that can be created and deployed inside k8s cluster. It is an encapsulation including a container (or several in some cases) typically, a network interface with an IP address and storage resources. One pod represents one instance of an application, and it can be scaled with other k8s objects, which will be introduced below.

ReplicaSet

ReplicaSet ensures the specific number of pods are running in a k8s cluster. It keeps monitoring the number of running pods and compare it with that in the configuration. If the two number are not matched, a corresponding action, scaling up or down, will be taken to make them equal. ReplicaSet is rarely used solely, usually a high-level object **deployment** is configured to provide declarative updates to pods using ReplicaSet.

Deployment

Deployment, as its name says, is a k8s object to deploy applications along with many useful features on a k8s cluster. For example, it provides a deployment strategy **RollingUpdate**, which updates application instances one-by-one so that users are not affected during the update. In addition, it manages deployment versions and

allows operators to switch to any previous version if a new deployment encounters problems. As a basic function, deployment can also change the configurations of a update, e.g. updating the number of pods using ReplicaSet, etc.

Service

As the previous description, **ReplicaSet** and **Deployment** maintain a specific number of pods and dynamically scale up or down pods if some pods are down or other problems occur. However, there is no guarantee that the newly spawned pods remain the same IP addresses as before. Assume there are some "users" interacting with some pods, and the problem is how "users" know and track these pods after the pods are restarted by k8s ReplicaSet or Deployment? A k8s object **Service** is used to solve the problem. A k8s service is an abstraction which defines a logical set of pods. It uses labels to define a set of pods to which the same labels are attached. Thanks to k8s service, "users" now only need to know the service, and not to be aware of any changes on the backend pods.

When a service is needed to be exposed to external, **ServiceTypes** should be specified. **ClusterIP** as the default service type, is used to expose a service with a cluster-internal IP. **NodePort** can expose a service to each node's IP at a static/dynamic port, which ranges within 30000-32767 by default. **LoadBalancer**, which may be provided by cloud providers, exposes a service to a public IP using a load balancer.

2.9 Summary

In this chapter, we introduced the background information and important concepts that are useful to understand the reminder of this thesis. First, a brief introduction to cloud computing and edge computing was given, highlighting also what are advantages and disadvantages of the two approaches. After, we discussed all the machine learning concepts that are relevant for this thesis, such as training inference, AI models, model serving, as well as the frameworks used in the experimental part of this work. Finally, container virtualization, container orchestration, and their usage to achieve microservices-based deployments were also discussed.

3 Methodology

This chapter provides the details of the methodology used in the experiments during our study. Specifically, we provide additional information about the usage of the AI models used in this work, the set up of a Kubernetes cluster on cloud and single-board computer, and how to provide microservice-based AI model serving through Tensorflow Serving and Clipper. Finally, we describe what evaluation method is used in order to measure the performance of the AI model serving.

3.1 Docker

Due to the fact that in our experiments all the AI serving tasks are performed through microservices, the installation of a container runtime engine, with a configuration suitable for our study's purposes, is the first step to accomplish. In our setup, Docker [54] is used because its wide popularity and adoption in the context of microservices-based platforms. In addition, we create a Docker Hub account to use its container image repository.

Install Docker

All the required steps needed to install Docker can be found in [64]. A successful Docker installation can be verified through the following command:

```
$ docker -v
Docker version 18.09.2, build 6247962
```

Register Docker Hub account

The Docker hub account can be registered at [59], and through the account one or more Docker image repositories can be created to store and download different versions of docker images created by the user.

3.2 Acquire AI trained models

In this section, the methods to acquire AI trained models used for our scope are given. Three models are used in our experiments. The first one is SSD inception v2 (COCO), a Tensorflow object detection model that was trained with the data from COCO [65]. A version of its trained model is available from Tensorflow Github [66]. The other two models are logistic regression and random forest, which are used to recognize handwriting digits from MNIST database [67]. For both of these two models, a small training job is needed.

3.2.1 SSD inception v2 (COCO)

This model is used only in Tensorflow Serving in our experiments. In this section, we provide the necessary steps for setting up the model to work. Before acquiring the model, a working directory **thesis** is created:

```
$ cd ~
$ mkdir thesis
$ export WORK_DIR=~/thesis
```

Next, we download the model from [66]. Scroll down to the section **COCO-trained models**, click the link **ssd_inception_v2_coco**, and save the file **ssd_inception_v2_coco_2018_01_28.tar.gz** (the file name may be changed later) to local directory, e.g. ~/Downloads. After that extract it into the working directory:

```
$ tar xvzf ~/Downloads/ssd_inception_v2_coco_2018_01_28.tar.gz -C $WORK_DIR
```

Once done this, we can see the following files/directories:

```
$ cd $WORK_DIR/ssd_inception_v2_coco_2018_01_28
$ ls
checkpoint frozen_inference_graph.pb model.ckpt.data-00000-of-00001
model.ckpt.index model.ckpt.meta pipeline.config saved_model
```

The model we will use is in a sub-directory called **saved_model**, which includes a serialized Tensorflow SavedModel (saved_model.pb) and the variables directory (variables).

```
$ cd saved_model
$ ls
saved_model.pb variables
```

Before saved_model.pb and variables can be used, it is recommendable to create a new directory called **1** – basically using the same name of the model version number – and move the serialized model and related variables to this sub-directory **1**:

```
$ mkdir 1
$ mv saved_model.pb variables 1
```

Once executed these preliminary tasks, the model is ready to be loaded and used by Tensorflow Serving.

3.2.2 Logistic regression and random forest

The models of logistic regression and random forest are acquired from scikit-learn with python script. To train the models, an appropriate set of data is needed. In our experiments, MNIST handwriting digits database [67] is chosen and downloaded

(from OpenML [68]) for this purpose:

```
from sklearn.datasets import fetch_openml
X, y = fetch_openml('mnist_784', version=1, return_X_y=True)
```

For the training phase, only the first 10000 samples are used to the train models – instead of 60000 for saving some time in the training phase –, and the last 20 samples are used to verify if the model training has been successful. It is worth highlighting that the data is not shuffled, in order to assure that the same trained model is obtained for each run of the script.

```
train_samples = 10000
test_size = 20
X_train = X[0:train_samples, :]
y_train = y[0:train_samples]
X_test = X[-test_size-1:-1, :]
y_test = y[-test_size-1:-1]
```

In order to normalize the data, we execute the following commands:

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

After pre-processing the data before the training phase, the model of logistic regression is trained:

```
lr = LogisticRegression(C=50. / train_samples, multi_class='multinomial',
penalty='l1', solver='saga', tol=0.1)
lr.fit(X_train, y_train)
```

In order to verify the trained model, we run:

```
print("Inference score: %.4f" % lr.score(X_test, y_test))
print("LR Output: ", lr.predict(X_test))
```

The printed output is:

```
Inference score: 0.7500
LR Output: ['7' '3' '6' '6' '0' '1' '7' '8' '4' '5' '6' '7' '8' '4' '0' '1' '2' '3' '4' '5']
```

Both the inference score, as well as the predicting results are reasonable. We repeated a similar process for the Random Forest model:

```
rf = RandomForestClassifier(n_estimators=500, min_samples_split=5)
rf.fit(X_train, y_train)
print("Inference score: %.4f" % rf.score(X_test, y_test))
print("RF Output: ", rf.predict(X_test))
```

The printed output is:

```
Inference score: 0.9500
RF output: ['7' '2' '6' '6' '0' '1' '2' '3' '4' '5' '6' '7' '8' '9' '0' '1' '2' '3' '4' '5']
```

It is worth pointing out that our goal is to obtain valid trained models, and assure that the same trained models are acquired for each run of the script. The accuracy of the trained model is a control variable and remain the same in all the experiments.

3.3 Setup of AI serving systems

This section gives information about the setup and the use of the two AI serving systems employed in our work: Tensorflow Serving and Clipper. Both tools can execute AI model serving operations, based on given models.

3.3.1 Tensorflow Serving setup

Tensorflow Serving provides an easy way to establish its setup. In particular, in the Tensorflow Serving docker image, two ports are exposed: port 8500 exposed for gRPC [70] and port 8501 for REST API. In our experiments, only the REST API is used. We need to mention that the UP board does not support some extensions (e.g. AVX [69], AVX2) to the x86 instruction set architecture, however, these extensions are used in the Tensorflow official docker image, i.e. **tensorflow/serving**. In order to overcome such limitation and make possible running Tensorflow on the UP board, we create a customized Tensorflow docker image that does not include the aforementioned extensions. To compare the performances of AI serving on cloud and on edge fairly, the same image will be used for both deployment environments.

To generate this customized Docker image, we create the following **Dockerfile**:

```
$ cd $WORK_DIR
$ vim Dockerfile
```

The content of the Dockerfile is shown below. Some lines of codes are referred from [71].

```
1  FROM ubuntu:16.04
2
3  RUN echo "deb [arch=amd64] http://storage.googleapis.com/tensorflow-
       serving-apt stable tensorflow-model-server tensorflow-model-server
       -universal" \
4      | tee /etc/apt/sources.list.d/tensorflow-serving.list && \
5      apt-get update && apt-get install -y curl && \
6      curl https://storage.googleapis.com/tensorflow-serving-apt/
           tensorflow-serving.release.pub.gpg | apt-key add - && \
7      apt-get update && apt-get install tensorflow-model-server-
           universal
8
9  EXPOSE 8500
10 EXPOSE 8501
11
12 ENV MODEL_BASE_PATH=/models
13 RUN mkdir -p ${MODEL_BASE_PATH}
14 ENV MODEL_NAME=model
15
16 RUN echo '#!/bin/bash \n\n\
17 tensorflow_model_server --port=8500 --rest_api_port=8501 \
18 --model_name=${MODEL_NAME} --model_base_path=${MODEL_BASE_PATH}/${
       MODEL_NAME} \
19 "$@"' > /usr/bin/tf_serving_entrypoint.sh \
20 && chmod +x /usr/bin/tf_serving_entrypoint.sh
21
22 ENTRYPOINT ["/usr/bin/tf_serving_entrypoint.sh"]
```

From the Dockerfile, line 1 shows that the image is based on Ubuntu 16.04. Line 3–7 give instructions to install Tensorflow ModelServer [72]. The key change in this custom docker image from the Tensorflow official one is on line 7. We use **tensorflow-model-server-universal**, which is compiled with basic optimizations without platform specific instruction sets, instead of tensorflow-model-server that includes more optimizations for machine learning computation [72]. Line 9–10 exposes gRPC service (port 8500, not used) and HTTP service (port 8501). The rest of the lines are to set environment variables and start tensorflow_model_server.

Once finished the Dockerfile for the customized image, we can build the Docker image specifying repository, image name, and the directory of the build context.

```
$ docker build -t boteliu/tensorflow_serving_universal .
```

Option **-t** is used to set a tag to image name, which is usually set as <repository_name>/<image_name>. In order to verify that the image is built successfully, we run:

```
$ docker image ls
REPOSITORY TAG IMAGE ID CREATED SIZE
boteliu/tensorflow_serving_universal latest 801d4f1c06b9 21 seconds ago 294MB
ubuntu 16.04 a3551444fc85 7 days ago 119MB
```

From the command results, both the image tensorflow_serving_universal and the base image Ubuntu 16.04 were created. However, the Tensorflow serving image created does not contain any models. Therefore we need to copy a Tensorflow model to a docker container started by the image and commit the container to obtain a working Tensorflow model server image with the given model:

```
$ docker run -d –name serving_base boteliu/tensorflow_serving_universal
$ docker cp $WORK_DIR/ssd_inception_v2_coco_2018_01_28/saved_model
serving_base:/models/universal_ssd_inception_v2_coco
$ docker commit –change "ENV MODEL_NAME
universal_ssd_inception_v2_coco" serving_base
boteliu/universal_ssd_inception_v2_coco
```

In order to verify the generated images, we run:

```
$ docker image ls
REPOSITORY TAG IMAGE ID CREATED SIZE
boteliu/universal_ssd_inception_v2_coco latest 9db5f78b7daa 50 seconds ago
397MB
boteliu/tensorflow_serving_universal latest 801d4f1c06b9 21 seconds ago 294MB
ubuntu 16.04 a3551444fc85 7 days ago 119MB
```

The command output shows one new image **boteliu/universal_ssd_inception_v2_coco** which includes the model **SSD inception v2 (COCO)**. The Tensorflow version can be seen by running a disposable container:

```
$ docker run –rm -i -t 9db5f78b7daa –version=true
TensorFlow ModelServer: 1.13.0-rc1+dev.sha.f16e777
TensorFlow Library: 1.13.1
```

With the current setup, we can run the Tensorflow Serving with the newly built custom image:

```
$ docker run -d 9db5f78b7daa
d59a0379fd1c8664eae2de66d57386bacff868d2b9590c05d5a7e31dad44047f
```

Next, we check the docker logs created by the Tensorflow model server (unimportant logs, timestamp, file names and lines are omitted as “...”):

```
$ docker logs d59a0379fd1c
... Building single TensorFlow model file config: model_name: universal_ssd_inception_v2_coco model_base_path: /models/universal_ssd_inception_v2_coco
...
... Loading servable version name: universal_ssd_inception_v2_coco version: 1
...
... Reading SavedModel from: /models/universal_ssd_inception_v2_coco/1
...
... Your CPU supports instructions that this TensorFlow binary was not compiled to use: SSE4.1 SSE4.2 AVX AVX2 AVX512F FMA
...
... Successfully loaded servable version name: universal_ssd_inception_v2_coco version: 1
... Running gRPC ModelServer at 0.0.0.0:8500 ...
... Exporting HTTP/REST API at:localhost:8501 ...

... RAW: Entering the event loop ...
```

According to the logs, it can be observed that the model **SSD inception v2 (COCO)** with version **1** is successfully loaded and read, and the gRPC and HTTP services are up for receiving requests. Also note that the specific instructions (SSE4.1, SSE4.2, AVX, AVX2, AVX512F, FMA) are not compiled in the image, enabling it to run on a UP board.

Then we tag the created image with a specific prefix and push the image to the Docker hub:

```
$ docker tag boteliu/universal_ssd_inception_v2_coco
docker.io/boteliu/universal_ssd_inception_v2_coco
$ docker push docker.io/boteliu/universal_ssd_inception_v2_coco
```

As last operation, it is recommended to clean the environment by running:

```
$ docker stop d59a0379fd1c
$ docker rm d59a0379fd1c
$ docker kill serving_base
$ docker rm serving_base
```

3.3.2 Clipper setup

This section provides the information for explaining setup and usage of Clipper for the purpose of our experiments. We write a python script in order to facilitate the setup and configuration of Clipper. In addition, two additional docker hub repositories will be created to push and download model images to be used through Clipper.

Before the actual setup of Clipper, a virtual environment creation was strongly recommended. A virtual environment tool creates isolated python virtual environments and keep dependencies for different projects. In our setup, Anaconda [73] is used for this purpose. Apart from providing virtual environments, Anaconda can be also used as a python package manager and a data science platform.

Install Anaconda

A guide of Anaconda installation is given at [74]. After the installation, verify the successful installation of the tool:

```
$ conda -V
conda 4.5.12
```

Create virtual environment

A virtual environment called 'myenv' with python version 3.6 is created and activated:

```
$ conda create -n myenv python=3.6
$ conda activate myenv
```

Install python packages

In the newly created virtual environment, we install all the needed python packages through pip or conda as shown below:

```
$ pip install clipper-admin==0.3.0
$ pip install cloudpickle==0.5.3
$ pip install sklearn==0.20.2
$ pip install docker==3.7.0
$ pip install kubernetes==8.0.1
$ pip install numpy==1.16.0
$ pip install requests==2.21.0
$ pip install scipy==1.2.0
```

Alternatively, packages can be installed with a text file (e.g. requirements.txt), where requirements.txt includes all packages to be installed.

```
$ conda install –file requirements.txt
```

Create Docker Hub repository

Two Docker hub repositories with the following names are created for Clipper, in order to push and download Docker images:
boteliu/logistic-regression-model
boteliu/random-forest-model

Changes to clipper-admin

In order to run Clipper, one line of code is changed in order to get a successful deployment. Below is the line 323 in clipper/clipper_admin/clipper_admin/kubernetes/kubernetes_container_manager.py:

```
if addr.type == "ExternalDNS":
```

where "ExternalDNS" is supposed to be replaced with "ExternalIP".

In addition, static nodePorts (**31337** and **31338**) are configured in query-frontend-service.yaml and mgmt-frontend-service.yaml in the directory clipper_admin/kubernetes, for a convenient firewall setup.

Besides, we can disable the caching for prediction results, by editing the file **clipper_admin.py** on line 32:

```
DEFAULT_PREDICTION_CACHE_SIZE_BYTES = 0
```

Docker DNS

Docker uses Google's public DNS (8.8.8.8 and 8.8.4.4) as its default DNS server. If such DNS setup is not working in your environment, it is required to add a customized DNS servers configuration to **/etc/docker/daemon.json**:

```
{
  "storage-driver": "overlay2",
  "live-restore": true,
  "dns": [
    "<Your DNS IP1",
    "<Your DNS IP2>"
  ]
}
```

Restarting Docker is needed in order to make the changes effective:

```
$ sudo service docker restart
```

Python script to run Clipper

Some extracts of the python script to run Clipper are provided below. The setup of the Docker Hub login is given as:

```
try:
    docker_client = docker.from_env()
    docker_client.login(username="boteliu", password="******")
except APIError as err:
    print(err)
```

Next, start Clipper with Kubernetes manager:

```
clipper_conn = ClipperConnection(KubernetesContainerManager())
clipper_conn.start_clipper()
```

Then, register two Clipper-defined applications and expose its REST API for predicting requests:

```
clipper_conn.register_application(name="bote-mnist-logistic-
    regression-demo", input_type="integers", default_output="1",
    slo_micros=1000000)
clipper_conn.register_application(name="bote-mnist-random-forest-demo
    ", input_type="integers", default_output="1", slo_micros=1000000)
```

The parameter “name” is the unique name of the application and a REST API of POST /<name>/predict is exposed to serve prediction requests. The parameter “slo_micros” is the query latency objective for the application in microseconds. If the application fails to receive response from a model within the specified “slo_micros”, the value of the parameter “default_output” will be returned.

Then, define the two models with python closures (the context about this is given in section 3.2.2):

```
def logistic_regression_mnist(xs):
    return lr.predict(xs)

def random_forest_mnist(xs):
    return rf.predict(xs)
```

After their definition, deploy the defined models as:

```
python_deployer.deploy_python_closure(clipper_conn, name="logistic-
    regression-model", version=1, input_type="integers", func=
    logistic_regression_mnist, num_replicas=3, pkgs_to_install=['
    scikit-learn', 'numpy==1.16.*'], registry="boteliu")
python_deployer.deploy_python_closure(clipper_conn, name="random-
    forest-model", version=1, input_type="integers", func=
    random_forest_mnist, num_replicas=3, pkgs_to_install=['scikit-
    learn', 'numpy==1.16.*'], registry="boteliu")
```

The parameter “func” is the prediction function, representing the trained model in section 3.2.2. The parameter “num_replicas” configures the number of replicas in the kubernetes deployment. “pkgs_to_install” specifies the extra python packages needed to install, and “registry” refers to Docker container registry. The parameter “batch_size” is not assigned explicitly, a default value “-1” is given, meaning that Clipper will adaptively calculate the batch size for individual replicas of this model.

The last step is to link the AI models to the applications registered earlier:

```
clipper_conn.link_model_to_app(app_name="bote-mnist-logistic-
    regression-demo", model_name="logistic-regression-model")
clipper_conn.link_model_to_app(app_name="bote-mnist-random-forest-
    demo", model_name="random-forest-model")
```

After that, a prediction request to a registered application will be forwarded to the model that links to the application.

Once that all the steps in the script are executed and the deployment is finished, we may find that the pod of query-frontend occupies about 1000m (milicores) CPU even if there are no requests. This is a known bug of Clipper – as explained in here [75] – that the new release should fix, although the estimated influence of the bug is low when the CPU usage is not very high.

3.4 AI model serving on GKE

In this section, the information related to the setup and configuration of GKE is given. In addition, more details about the deployment of AI model serving on GKE using Tensorflow Serving and Clipper are provided.

3.4.1 GKE setup

In order to create a Kubernetes cluster with GKE, the following steps are needed:

- Create a GCP account.
- Install Google Cloud SDK.
- Install Kubernetes command-line tool (kubectl).
- Create a Kubernetes cluster with GKE.

Beside the preparation work above, some additional configurations are also needed.

Fetch credentials for a running cluster

To connect the GKE cluster from a machine, the credentials of the cluster can be obtained through the following command:

```
$ gcloud container clusters get-credentials <cluster-name> –zone <zone-name>
–project <project-name>
```

Switch cluster context

If more than one Kubernetes cluster are created, it is important to inspect what are the existing clusters, and then to switch to the one supposed to be used. In order to do this, we first display the list of the different contexts:

```
$ kubectl config get-contexts
```

Then, we display the currently used context:

```
$ kubectl config current-context
```

Finally, we select the context that we wish to use:

```
$ kubectl config use-context <my-cluster-name>
```

Check everything is fine

After switching to the right context, we can check the nodes belonging to that cluster (the GKE cluster name is **quickstart**):

```
$ kubectl get node
NAME STATUS ROLES AGE VERSION
gke-quickstart-default-pool-db2f7a4a-25mw Ready <none> 1h v1.11.8-gke.6
gke-quickstart-default-pool-db2f7a4a-6rgh Ready <none> 1h v1.11.8-gke.6
gke-quickstart-default-pool-db2f7a4a-g2t1 Ready <none> 1h v1.11.8-gke.6
gke-quickstart-default-pool-db2f7a4a-qf36 Ready <none> 1h v1.11.8-gke.6
```

Also, list all the k8s resources in the cluster:

```
$ kubectl get all
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service/kubernetes ClusterIP 10.0.0.1 <none> 443/TCP 74d
```

In the shown example, only one k8s service (default) is there.

3.4.2 Deploy AI serving with TensorFlow Serving on GKE

Figure 3 shows the architecture for the AI model serving deployed by Tensorflow Serving on GKE cluster. The service of the AI prediction is exposed by a type of k8s service called "LoadBalancer", which is supported by GKE. The "LoadBalancer" exposes the applications inside the k8s cluster to a public IP address.

In order to provide AI model serving with Tensorflow Serving on GKE cluster through Docker images, a YAML [53] file is written to configure k8s resources. The following code is used to create a k8s deployment with 8 pods containing the docker image **universal_ssd_inception_v2_coco** that was created in the previous section.

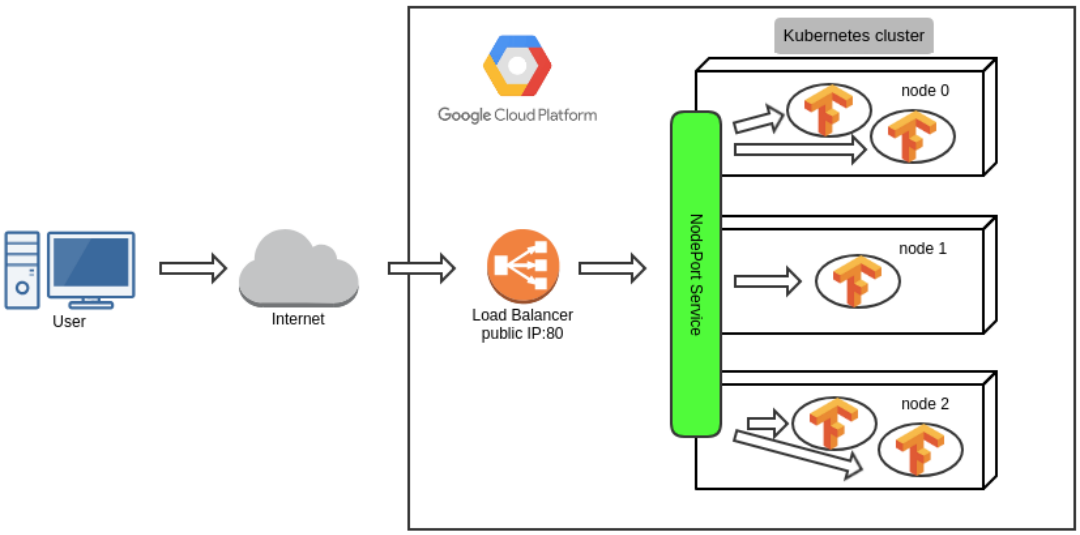

Figure 3: AI serving deployed with Tensorflow Serving on GKE cluster

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: ssd-inception-v2-coco
spec:
  replicas: 8
  template:
    metadata:
      labels:
        app: ssd-inception-v2-coco
    spec:
      containers:
      - name: ssd-inception-v2-coco
        image: docker.io/boteliu/universal_ssd_inception_v2_coco
        ports:
        - containerPort: 8501
```

The k8s deployment is running and monitoring the status of the pods. If one or more pods are down for a particular reason, the k8s deployment will create new pods to keep the number of replicas unchanged. For each pod created in the k8s deployment, a Docker image **boteliu/universal_ssd_inception_v2_coco** is downloaded from the Docker hub, and a container from that image is started in the pod. Note that port 8501 is exposed, meaning only HTTP service is provided.

The following YAML code is written to create a k8s service:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    run: ssd-inception-v2-coco
  name: ssd-inception-v2-coco-service
spec:
  ports:
  - name: http
    port: 80
    targetPort: 8501
  selector:
    app: ssd-inception-v2-coco
  type: LoadBalancer
```

A k8s service uses labels to associate the pods to which one or more of the labels are attached. When a k8s service receives a request, it forward the request to the associated pods. The "LoadBalancer" service configured above exposes the applications that are in the pods with a label "app: ssd-inception-v2-coco" to a load balancer with a public IP and port 80. When the load balancer receives a request, the request will be forwarded to the service and forwarded again to one of the pods that associate the service.

With the following instructions, we can create k8s resources through a YAML file including the k8s "Deployment" and "Service":

```
$ kubectl create -f conf.yaml
deployment.extensions/ssd-inception-v2-coco created
service/ssd-inception-v2-coco-service created
```

It is always recommendable to check the status of the deployment on the cluster:

```
$ kubectl get deploy
NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
deployment.apps/ssd-inception-v2-coco 8 8 8 8 46s
```

as well as the service on the cluster:

```
$ kubectl get svc
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service/kubernetes ClusterIP 10.0.0.1 <none> 443/TCP 74d
service/ssd-inception-v2-coco-service LoadBalancer 10.0.8.226 35.228.46.101
80:32499/TCP 45s
```

and the pods on the cluster:

```
$ kubectl get pod
NAME READY STATUS RESTARTS AGE
pod/ssd-inception-v2-coco-85f544f555-4ttc9 1/1 Running 0 45s
pod/ssd-inception-v2-coco-85f544f555-7g2q9 1/1 Running 0 45s
pod/ssd-inception-v2-coco-85f544f555-869xd 1/1 Running 0 45s
pod/ssd-inception-v2-coco-85f544f555-b69q6 1/1 Running 0 45s
pod/ssd-inception-v2-coco-85f544f555-q592q 1/1 Running 0 45s
pod/ssd-inception-v2-coco-85f544f555-qxmxb 1/1 Running 0 45s
pod/ssd-inception-v2-coco-85f544f555-rfp4h 1/1 Running 0 45s
pod/ssd-inception-v2-coco-85f544f555-s2psx 1/1 Running 0 45s
```

All the resources are created successfully, and the assigned public IP of the load balancer is 35.228.46.101.

3.4.3 Deploy AI serving with Clipper on GKE

Figure 4 shows the architecture of the AI model serving provided by Clipper on GKE cluster. The service of the query front-end is exposed by a type of k8s service called "NodePort", which exposes the applications inside cluster on each node's IP at a static/dynamic port (i.e. the NodePort – ports' number range between 30000 and 32767).

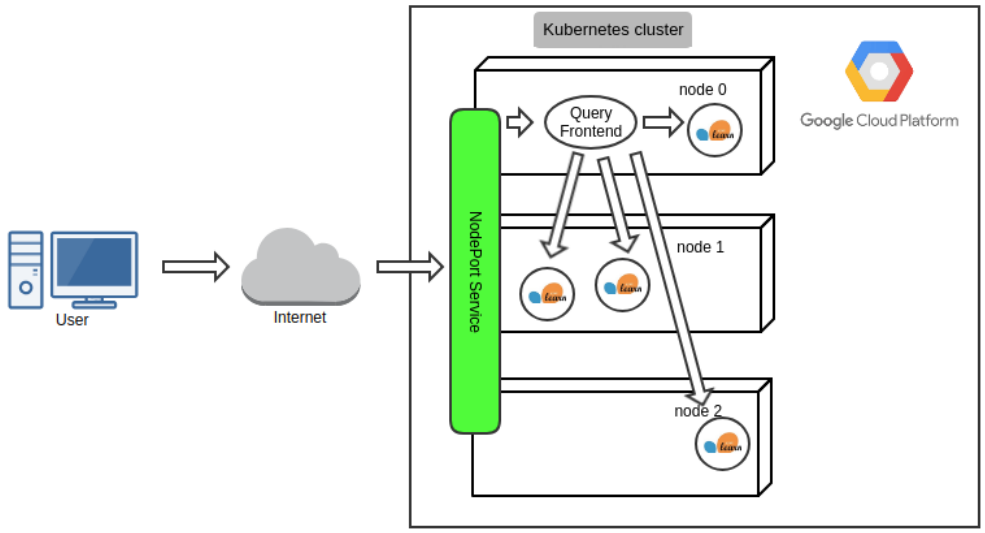

Figure 4: AI serving deployed with Clipper on GKE cluster

Next, we set the number of both model replicas to 3 and execute the python script already described in section 3.3.2. After running the script, we verify the execution and check the k8s deployment:

```
$ kubectl get deploy
NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
logistic-regression-model-1-deployment-at-0 3 3 3 3 1d
metrics 1 1 1 1 1d
mgmt-frontend 1 1 1 1 1d
query-frontend-0 1 1 1 1 1d
random-forest-model-1-deployment-at-0 3 3 3 3 1d
redis 1 1 1 1 1d
```

as well as the k8s services:

```
$ kubectl get svc
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubernetes ClusterIP 10.0.0.1 <none> 443/TCP 1d
metrics NodePort 10.0.5.188 <none> 9090:32537/TCP 1d
mgmt-frontend NodePort 10.0.0.221 <none> 1338:31338/TCP 1d
query-frontend NodePort 10.0.4.8 <none> 1337:31337/TCP 1d
query-frontend-0 NodePort 10.0.5.175 <none> 7000:31713/TCP 1d
redis NodePort 10.0.3.44 <none> 6379:31665/TCP 1d
```

and the k8s pods:

```
$ kubectl get pod
NAME READY STATUS RESTARTS AGE
logistic-regression-model-1-deployment-at-0-5c6d8b48cf-c2scz 1/1 Running 1 16h
logistic-regression-model-1-deployment-at-0-5c6d8b48cf-cn7wv 1/1 Running 0 16h
logistic-regression-model-1-deployment-at-0-5c6d8b48cf-xnj8q 1/1 Running 0 16h
metrics-86cff74b9d-8rxkd 1/1 Running 0 16h
mgmt-frontend-5c768475d8-6lljf 1/1 Running 0 16h
query-frontend-0-d84d64dd6-ghffd 2/2 Running 0 16h
random-forest-model-1-deployment-at-0-d76f9959c-92nnd 1/1 Running 0 16h
random-forest-model-1-deployment-at-0-d76f9959c-m9kcg 1/1 Running 1 16h
random-forest-model-1-deployment-at-0-d76f9959c-rbqpf 1/1 Running 0 16h
redis-5cf6d8df45-twzrj 1/1 Running 0 16h
```

NodePort exposes services to the node ports between 30000 and 32767. However, these ports are closed on GCP by default. In order to be able to use these two ports, the following two firewall rules are created:

```
$ gcloud compute firewall-rules create query-frontend-node-port –allow tcp:31337
$ gcloud compute firewall-rules create mgmt-frontend-node-port –allow tcp:31338
```

3.5 AI serving on UP board

This section provides details about the setup of the Intel UP board and the method to deploy AI model serving on this board with Tensorflow Serving and Clipper. As k8s cluster abstracts (virtual) resources on OS into logical resources on a cluster, the setup and the configurations of the AI model serving and of the k8s cluster running

on GKE and UP board are very similar. The details of such setup and configurations can be found in section 3.4 and will not be repeated in this section. The minor differences could be due to the specific features provided by different k8s providers. For example, GKE provides the feature of "LoadBalancer" service, while this feature is not supported by microk8s, which is a tool used for facilitating the creation of a k8s cluster. In addition, it is important to switch to the right k8s context when configuring different k8s clusters.

3.5.1 UP board setup

The OS installed on the board is Ubuntu 16.04. In order to have more computational resources available on the single-board computer, the Ubuntu GUI is disabled:

```
$ sudo service lightdm stop
```

In addition, microk8s is chosen to create a k8s cluster on the board because it does not require a hypervisor. Install microk8s on the board:

```
$ sudo snap install microk8s –classic
```

Next, start the microk8s cluster:

```
$ sudo microk8s.start
```

We also enable some specific features in microk8s (e.g. dns and dashboard), which are not loaded by default:

```
$ microk8s.enable dns dashboard
```

After that, we check the status of the microk8s nodes:

```
$ microk8s.kubectl get node
NAME STATUS ROLES AGE VERSION
n222 Ready <none> 15d v1.14.1
```

In order to allow the connection to the k8s cluster from outside the UP board, we need to edit **/var/snap/microk8s/current/args/kube-apiserver** and set

```
–insecure-bind-address=<Your IP>
```

For testing purpose, <Your IP> could be put 0.0.0.0 temporarily. Then, a restart is needed:

```
$ microk8s.stop
$ microk8s.start
```

3.5.2 Deploy AI serving with TensorFlow Serving on edge

Figure 5 shows the architecture of the AI model serving deployed by Tensorflow Serving on the UP board. The board and users are connected through Ethernet, and the Tensorflow model serving is exposed by the k8s “NodePort” service. The k8s YAML configuration file used in this case is very similar as the one described

in section 3.4.2, with the exception that a “NodePort” service is used instead of “LoadBalancer”.

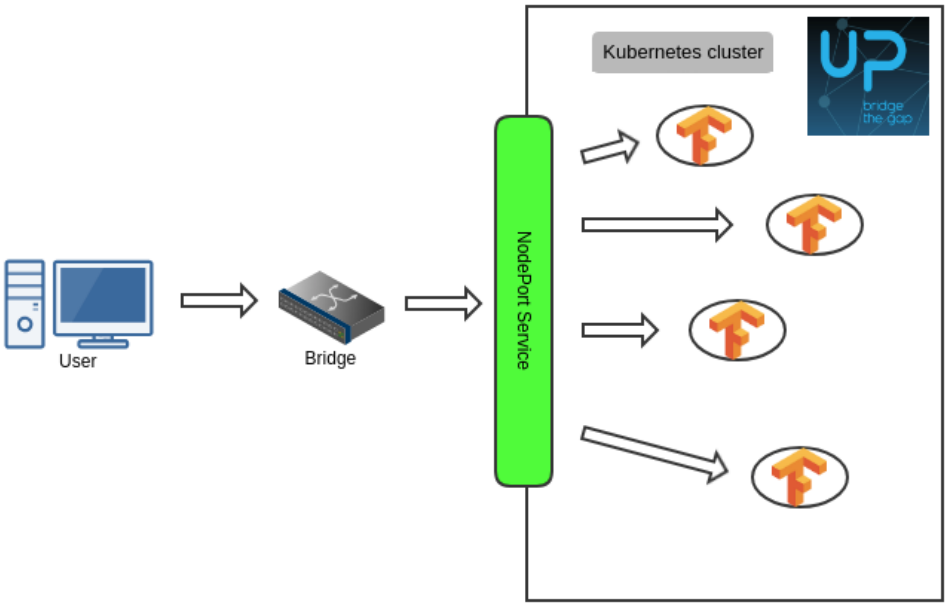

Figure 5: AI serving deployed with Tensorflow Serving on edge

3.5.3 Deploy AI serving with Clipper on edge

Figure 6 shows the architecture of the AI model serving performed with Clipper on the UP board. A bridge is used to connect users and the board. The Clipper query front-end is exposed by k8s “NodePort” service. The python script to configure Clipper is the same as the one described in section 3.3.2. It is important to remember to switch to the right k8s context in order to make the system working properly.

3.6 Performance measurement method

This section describe the methods used to measure and evaluate the performance of the AI model serving, which uses the setup and configurations introduced in the section 3. The measurement can be divided into two parts: preparation of the requests payload and the actual performance measurement of the AI model serving.

3.6.1 Prepare request payload

The payload of requests differs depending on the different AI models that we consider. The model of SSD inception v2 (COCO) [45] is used to detect objects and expects some images as input, while for logistic regression and random forest, MNIST handwriting images [67] are chosen.

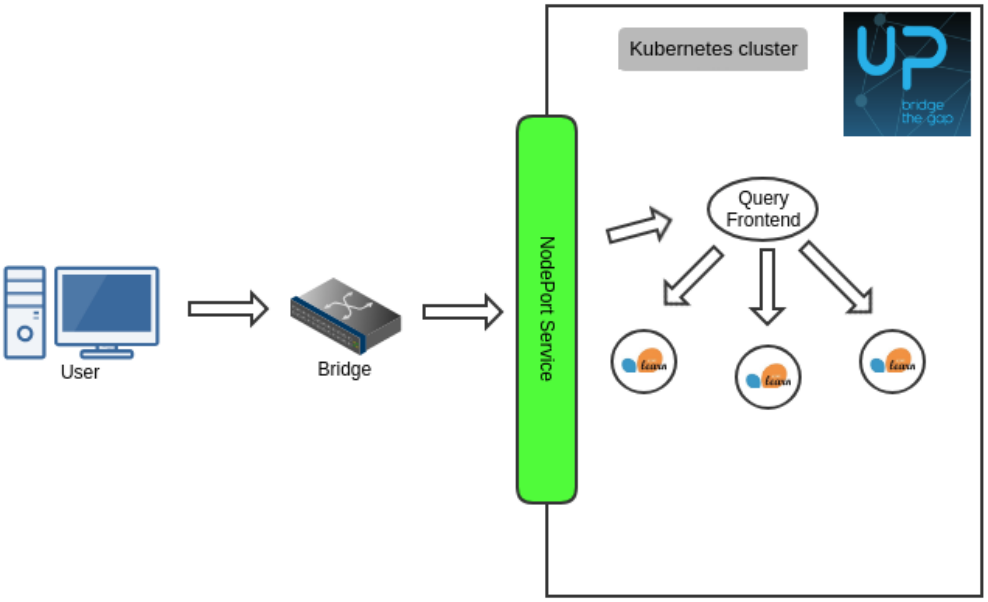

Figure 6: AI serving deployed with Clipper on edge

CIFAR-10

CIFAR-10 [76] includes 60000 (32x32 resolution) color images in 10 classes, and it is chosen as the requesting input to the model of SSD inception v2 (COCO). It can be downloaded from [76], but only the first 10000 tiny images (in file data_batch_1) are used in our requests.

Common Object in Context (COCO)

The COCO is a large-scale object detection, segmentation, and captioning dataset [65]. Unlike CIFAR-10, the sizes of COCO images that we use are much larger. Most of the COCO images we use (COCO 2017 val images) have different sizes. Figure 7 shows the distribution of the image resolutions.

MNIST

MNIST is a database of handwriting digits images [67], and these images are only used for the models of logistic regression and random forest in our experiments. Each MNIST image is a 28x28 tiny handwriting image, and these images can be downloaded from OpenML [68] with the following python script:

```
from sklearn.datasets import fetch_openml
X, y = fetch_openml('mnist_784', version=1, return_X_y=True)
```

where if **return_X_y** is True, the function returns (data, target) instead of a Bunch object.

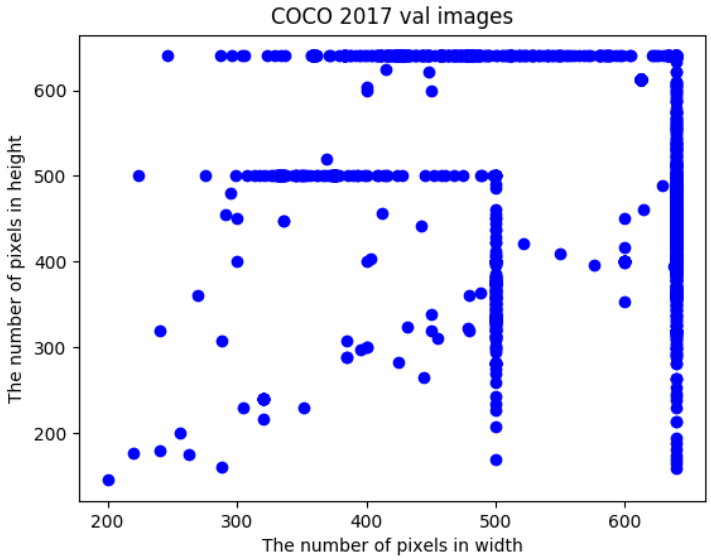

Figure 7: The distribution of the COCO image resolutions

3.6.2 Measure performance of serving

Figure 8 shows the method to measure the performance of AI model serving. The whole process consists of two sequential steps:

(1) Prepare request data and push them in a synchronized queue.

(2) Start multiple threads, each of which fetch data from the queue and send requests to AI serving until the queue is empty.

Step (1) creates the conditions for executing the actual measurement (which is happening in (2)). Step (2) is the actual task for performing the measurement. In step (2), requests are sent to the AI model serving system in parallel, and counters and timers are set up to store the measurement results. It is important to clarify that the time spent on Step (1) does not influence the measurement results. A python script implementing the whole process (using image data of CIFAR-10) is provided in the appendix A.

The most important performance metrics that we are interested are throughput and response time of the AI model serving. At the same time, CPU and memory utilization of our setup are also worth of attention. Before running the performance measurements, a moderate amount of requests are sent to the AI model serving to warm up the system, so as to achieve performance results close to its working state.

Throughput and response time

The throughput is expressed by the total number of successful response to requests over the time interval of the step (2). For the response time, a timer is set for each

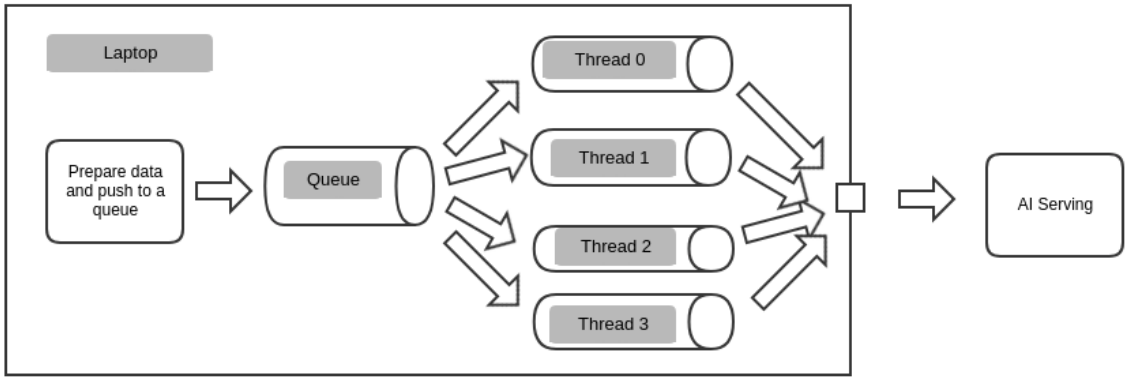

Figure 8: The method to measure the performance of AI model serving

request to measure the interval between the time the request is sent and the response is received. A statistics tool is used to calculate the mean and the standard deviation of the response time with the data from all timers.

CPU and memory utilization

The CPU and memory utilization on the serving side are coarsely measured with heapster [77] installed on k8s cluster:

```
$ kubectl top node
```

3.7 Summary

This chapter introduced the methodology used to setup the experimental environments needed to perform the empirical analysis of the thesis. In the chapter, we covered different aspects including the setup of the underlying Edge AI and Cloud AI environment (e.g. Docker, k8s, GKE, etc.), the model serving systems (e.g. Tensorflow Serving and Clipper), as well as additional information useful for reproducing the working experimental setups. Finally, we discussed the methodology used to measure the performance of both Cloud AI and Edge AI deployments.

4 Results and Analysis

This chapter presents the results collected from the different system setup that are described in Chapter 3. The results include performance metrics of the AI model serving deployed on cloud and on edge separately and a comparison is made to see the advantages and disadvantages of the two deployment options.

The performance metrics that we are interested in are the throughput of the AI model serving, the response time to prediction requests, and the resources (CPU and memory) consumed by model serving in the distinct deployment environments.

Three AI models are used in our performance evaluation:

- SSD inception v2 (COCO)
- Logistic regression
- Random forest

The model **SSD inception v2 (COCO)** is used for object detection, and the serving of this model is operated by Tensorflow Serving. The models of **logistic regression** and **random forest** are used to recognize handwriting digits, and their serving are provided by Clipper. It is worth noting that the trained models are control variables in our study, i.e. the trained models are kept unchanged throughout an experiment. As a consequence, the prediction accuracy of the same trained models has no differences (with the same testing data), and therefore it is not necessary to show and compare prediction accuracy in the results.

The results are collected from two deployment environments: cloud and edge. For cloud environment, a k8s cluster consisting of two nodes is created on GKE in the zone *europe-north1-b*. The edge environment is represented by an Intel UP board connected to the served users through LAN network.

Table 1: The resource configurations of two deployment environments

Env	Hardware	Nodes	Network	CPU	Memory
Cloud	GKE	2	Internet	2*2 (2.0GHz, up to 2.7–3.5GHz)	2*2.25GB
Edge	UP board	1	LAN	4*1 (1.44GHz, up to 1.92GHz)	4GB

Table 1 shows the resources configuration of the two deployment environments. **Cloud** uses a k8s cluster on GKE, and it uses two nodes with powerful CPUs. There is a little compensation (0.25GB for each node) made for the memory considering the extra overhead of the OS in two nodes. The **Edge** environment consists of an Intel UP board with 4 cores (less powerful than the CPUs of the GKE setup) and 4GB RAM memory. It is worth highlighting that the CPUs in the cloud setup are not only faster, but also newer and support more instruction set extensions (than UP board) to optimize the computation of model serving. Table 2 shows the basic

overhead to start up the two environments, and it can be observed that a decent amount of resources are used even if there is no yet any particular workload running.

Table 2: The basic overhead to initialize the environments

Hardware	CPU	Memory
GKE	3%, 5%	35%, 60%
Edge	9%	31%

In order to run the experiments, we write a python script to measure the performance metrics. In the script, threads are used to simulate users, and we want to see the performances for different number of threads. In addition, the number of k8s pods – each of which represents one back-end application – is also a variable in our measurements. When more than one pod is configured, they are allocated to different nodes on GKE cluster to make use of the resources of both nodes.

4.1 Model SSD-inception-v2 (COCO)

This section shows the results of the performance metrics of the AI model serving with **SSD inception v2 (COCO)**. The payload of the HTTP predicting requests uses the images from CIFAR-10 and COCO. CIFAR-10 is a database consisting of 32x32 colour images in 10 classes, while COCO has images with much higher resolutions.

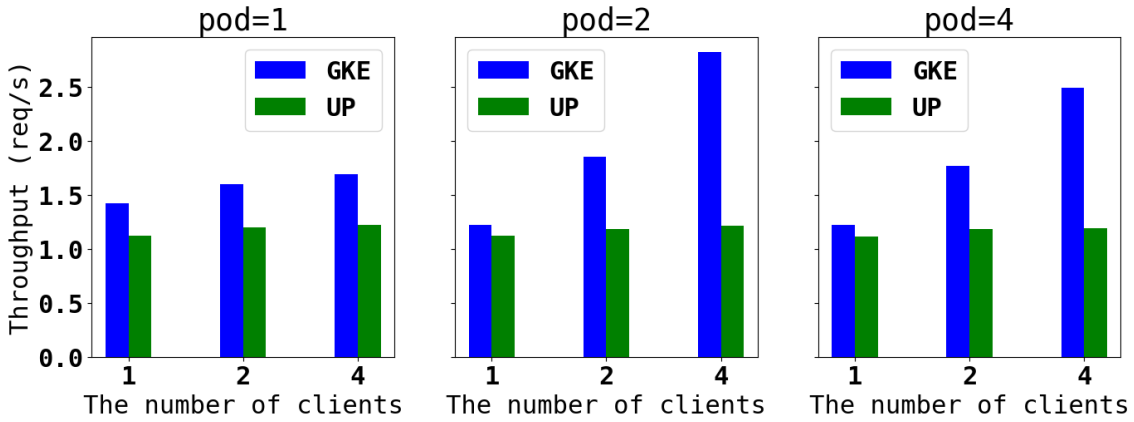

Figure 9: Throughput of the AI model serving with the model "SSD inception v2 (COCO)" and request payload from CIFAR-10

Figure 9 shows the throughput of the AI model serving with model **SSD inception v2 (COCO)** and table 3, 4, 5 and 6 show the CPU and memory usage for the same set of experiments. In the measurements, the HTTP requests use images from CIFAR-10 as payload. According to the results, both throughput of GKE cluster and UP board are quite low, i.e. less than 3 requests per second, and it can be

deduced that this model serving requires very high computation resources, which can be verified by the high CPU utilization shown in table 3 and 5. In addition, it can be observed that the throughput of the GKE cluster outperform the one of the UP board in all cases, and the reason is that the CPUs of GKE are more powerful and can process compute-intensive requests faster. Besides, the throughput of the serving on the UP board remains almost constant regardless of the number of users or k8s pods. This result can be explained by the fact that the CPU usage of the UP board is close to 100% even for one user and one pod, so the performance cannot be obviously improved as the increase of the users and k8s pods due to the shortage of CPU resources. In contrast, as the number of users increases, the throughput of the model serving on GKE cluster is higher, and its advantage over the UP board is greater. The increased number of users boosts the CPU usage on the GKE cluster, and the model serving can process more requests from multiple users and reduce the idle time waiting for requests. In addition, the throughput of model serving on GKE cluster is also improved when the number of k8s pods is increased from 1 to 2. Such improvement relates to the fact that the two k8s pods can make use of the resources in both nodes instead of one. However, the performance increase is not persistent when the number of k8s pods reaches 4, and the reason is that the increase of the k8s pods cannot provide more capacity to perform computation-intensive tasks. In addition, a memory shortage occurs, as shown in table 4 – the memory usage of both nodes is 100% when 4 pods are created.

Table 3: CPU utilization of the two nodes in GKE cluster with model "SSD inception v2 (COCO)" and request payload from CIFAR-10

P / T	1	2	4
1	5%,86%	50%,31%	39%,39%
2	5%,98%	55%,58%	64%,52%
4	5%,100%	82%,96%	95%,68%

Table 4: Memory utilization of the two nodes in GKE cluster with model "SSD inception v2 (COCO)" and request payload from CIFAR-10

P / T	1	2	4
1	50%,79%	78%,79%	100%,100%
2	50%,79%	78%,79%	100%,100%
4	50%,79%	78%,79%	100%,100%

Table 5: CPU utilization of UP board with model "SSD inception v2 (COCO)" and request payload from CIFAR-10

P / T	1	2	4
1	95%	95%	94%
2	99%	99%	99%
4	100%	100%	100%

Table 6: Memory utilization of UP board with model "SSD inception v2 (COCO)" and request payload from CIFAR-10

P / T	1	2	4
1	40%	53%	77%
2	40%	53%	77%
4	40%	54%	77%

Figure 10 shows the response time with the same setup mentioned above. The results show that the response time of the AI model serving on the UP board is longer than that on GKE cluster even if its network latency is lower, and the result is due to the bottleneck caused by the less powerful CPUs, which are the weakness of the UP board. However, UP board has its own merit that the standard deviation of the response time is smaller than GKE cluster, and the reason is that the network setup of the UP board makes it have a relatively stable network latency. In addition, from the results it can be observed that the waiting time of users is obviously longer in the case of UP board as the number of users increases. The cause of this result is that the CPUs on the UP board are overloaded and cannot serve more requests. Moreover, the increase in the number of k8s pods does not help on improving the response time for the UP board, because the overall processing capacity cannot be increased if the CPU usage are already saturated.

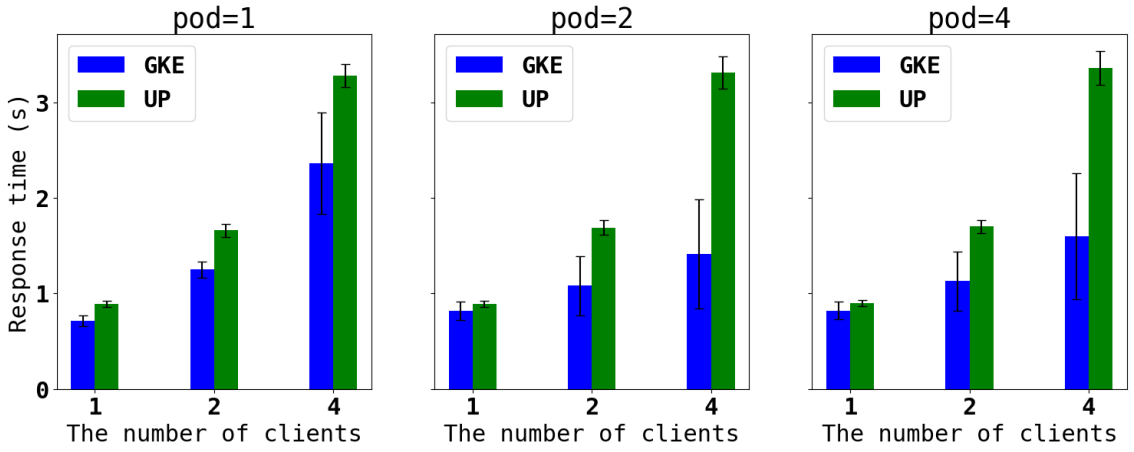

Figure 10: Response time of the AI model serving with the model "SSD inception v2 (COCO)" and request payload from CIFAR-10

Next, we look at the results with the same AI model, but the request payload are the images from COCO, which includes relatively bigger pictures. More details about the request payload are introduced in section 3.6.1. Figure 11 and Figure 12 show the results for this set of experiments. At the first glimpse, the overall throughput is quite low – about 1 request per second or lower – and it indicates the serving in this case is low-throughput and takes more time to process each request. The UP board has the better performance of both throughput and response time when the number of users is small, i.e. 1 and 2. It can be deduced that in this case the resources on the UP board are not yet saturated (also verified by table 9 and 10), and another reason for the UP board outperforming GKE could be due to the request payload. Comparing with CIFAR-10, the resolutions of the images from COCO are hundreds times larger, and these images generate more network latency in the GKE deployment, which may cause the opposite results in the cases of payload CIFAR-10 and COCO. However, when the number of users reaches 4, the performance of GKE

is better than the UP board. In this case, the CPU usage of UP board is not further increased (see table 9) and therefore its performance improvement is not obvious, but the CPU usage of GKE increases sufficiently (see table 7) as the increased users. Moreover, the results shows that the overall performance is not improved as the k8s pods increase and the reason is that, although the increase of pods (from 1 to 2 for GKE) boost the potential processing capacity, the workload is lower than reaching the processing limit. On the other hand, the reason may be the same as the situation using image from CIFAR-10 that the increased k8s pods (from 2 to 4 for GKE, or from 1 to 4 for UP board) cannot boost the speed to finish computing-intensive jobs.

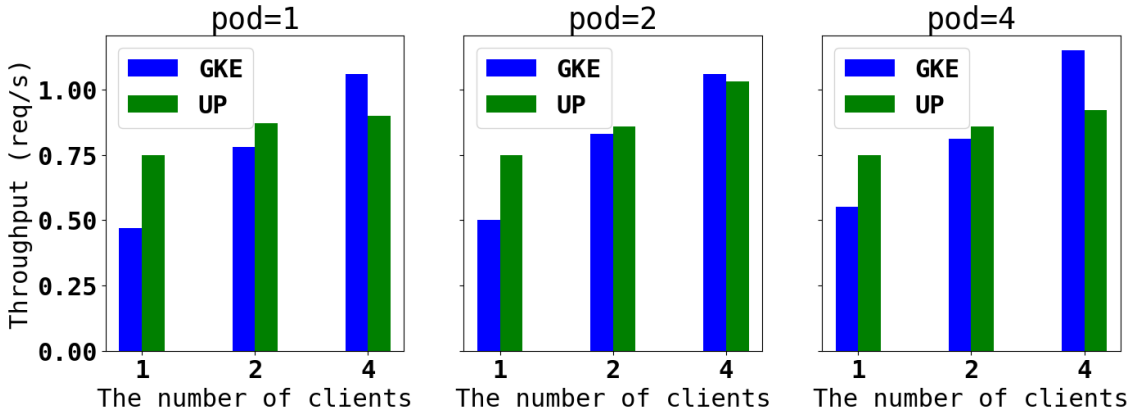

Figure 11: Throughput of AI model serving with the model "SSD inception v2 (COCO)" and request payload from COCO

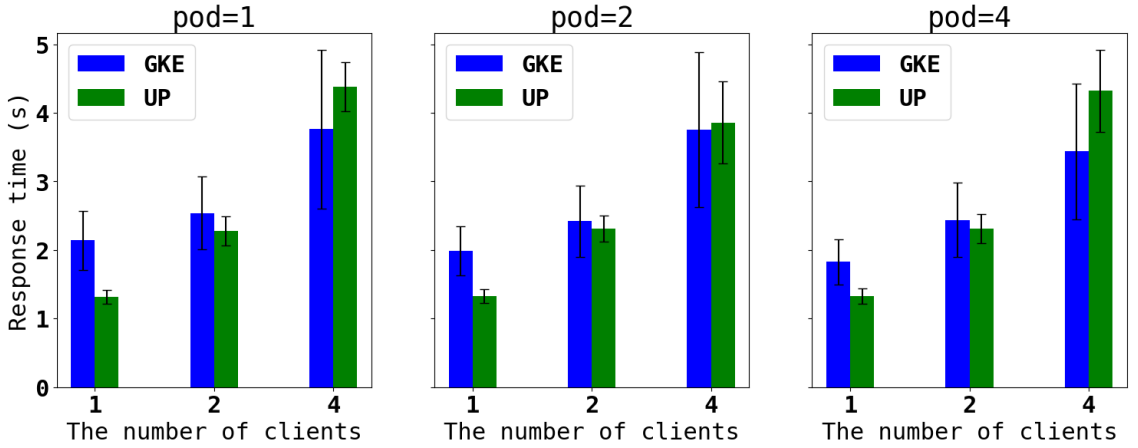

Figure 12: Response time of AI model serving with the model "SSD inception v2 (COCO)" and request payload from COCO

Table 7: CPU utilization of the two nodes in GKE cluster with model "SSD inception v2 (COCO)" and request payload from COCO

P / T	1	2	4
1	5%,32%	24%,14%	10%,30%
2	5%,57%	28%,33%	14%,44%
4	5%,69%	37%,49%	24%,59%

Table 8: Memory utilization of the two nodes in GKE cluster with model "SSD inception v2 (COCO)" and request payload from COCO

P / T	1	2	4
1	60%,63%	84%,75%	79%,100%
2	60%,67%	85%,75%	79%,100%
4	60%,72%	88%,75%	79%,100%

Table 9: CPU utilization of UP board with model "SSD inception v2 (COCO)" and request payload from COCO

P / T	1	2	4
1	70%	72%	71%
2	81%	80%	80%
4	82%	92%	87%

Table 10: Memory utilization of UP board with model "SSD inception v2 (COCO)" and request payload from COCO

P / T	1	2	4
1	44%	61%	82%
2	46%	62%	84%
4	48%	64%	85%

4.2 Model logistic regression

In this sections, the results of the model **logistic regression** are presented and discussed. The throughput and response time of the model serving in the case of logistic regression are shown in Figure 13 and 14. Unlike the model of object detection presented in the previous section, the results show that the model serving with logistic regression can reach a relatively high throughput (up to 100 requests per second). Based on the characteristics of the model and the achieved results, it can be deduced that the processing time for each request is low and the network latency may occupy the relatively big portion of the whole response time and therefore plays an important role in the performance results.

According to the results, the performance of the model serving on UP board is much better than the one of the GKE cluster in all conditions. The better CPUs on the GKE cluster does not generate an obvious benefit in this case, and the network latency is a more important factor to consider. The UP board and the users requesting the model serving are connected in the same LAN and the network latency is very low, explaining the superior performance of the UP board over the GKE cluster. As the number of the connected users increases, the model serving in both environments are improved. This enhanced performance occur even in the UP board, which can make use of its 4 cores to speed up its throughput. The increase of the k8s pods produces a negative influence on the UP board, because more resources are occupied, and any performance benefit cannot be observed due to the same reasons explained in section 4.1. It is worth noting that the (relative) standard deviation of the response time is

exceptionally high in GKE, and this can be explained by the fact that the effect of the network latency is much more obvious than the case of UP board.

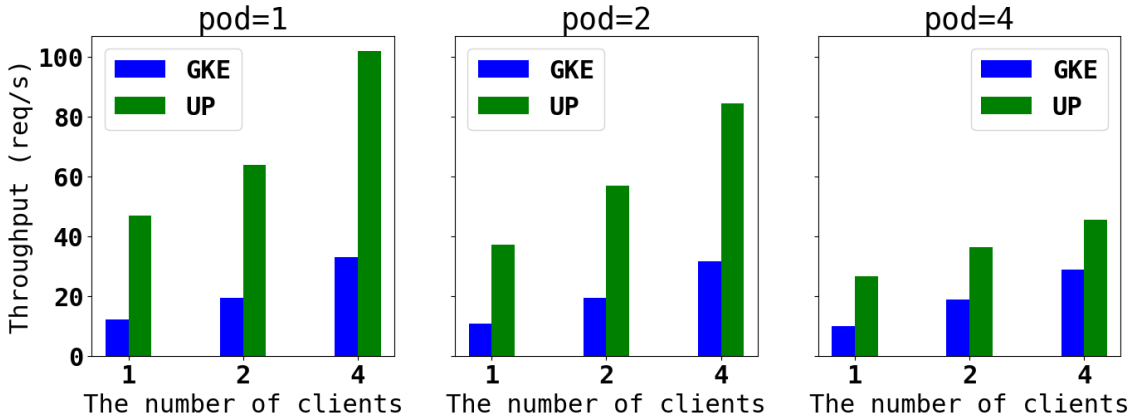

Figure 13: Throughput of AI model serving with the model "logistic regression" and request payload from MNIST

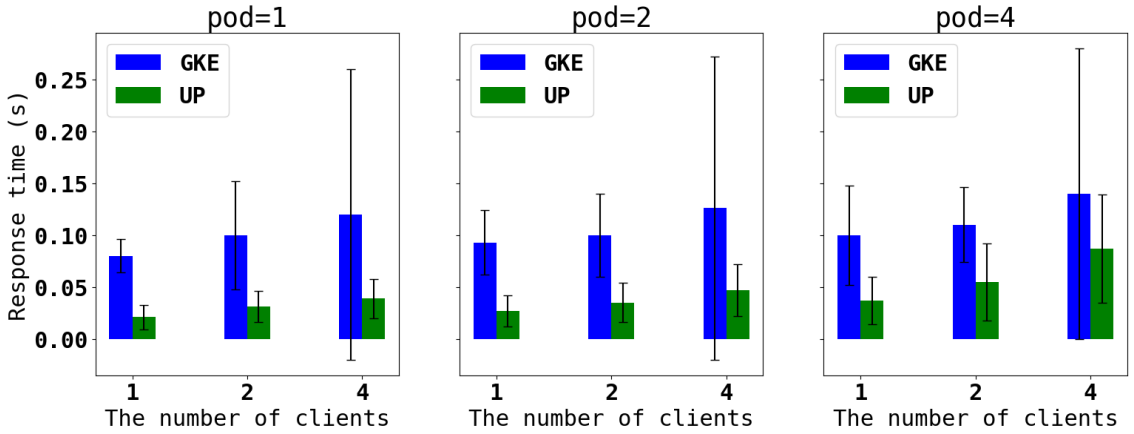

Figure 14: Response time of the AI model serving with the model "logistic regression" and request payload from MNIST

4.3 Model random forest

This section presents the results when the random forest model is used. Before commenting the achieved results, we clarify that when the number of k8s pods is set to 4, the model serving crashes due to the shortage of memory, and therefore results are not provided for that particular case. Figure 15 and Figure 16 show the results of throughput and response time in the case of one and two pods. The overall throughput is lower than the model of logistic regression, but higher than the object detection model. According to the results, the performance of the model serving on

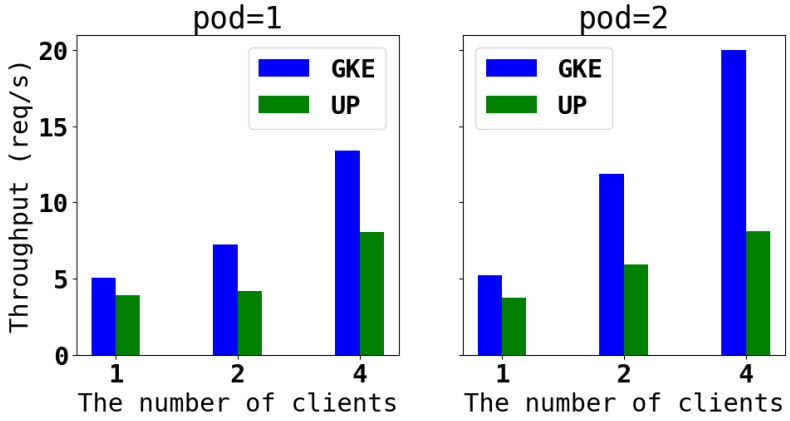

Figure 15: Throughput of the AI model serving with the model "random forest" and request payload from MNIST

GKE is better than the UP board. The powerful CPUs used in the GKE cluster may be the reason. The throughput of the model serving on UP board increases as the users increase, however, the growth is not as large as the one observable in the GKE cluster. When the number of k8s pods increases from 1 to 2, throughput and response time are both improved in GKE and UP board. This result can be explained by the additional resources (nodes and cores) allocated for accomplishing the serving tasks.

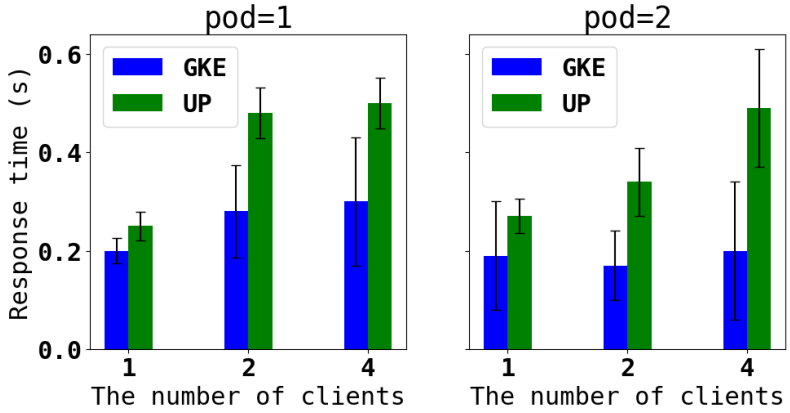

Figure 16: Response time of the AI model serving with the model "random forest" and request payload from MNIST

Caching of results

Finally, an experiment is conducted to investigate the effects of caching on the throughput of the AI model serving. To simplify the experiment, only the case of one user and one k8s pod is considered and Figure 17 shows such result. The throughput of the model serving on UP board is higher than the on achieved by the GKE, besides with a wide margin. In this situation, the time needed for processing the requests is

extremely low and the network latency is the factor that mostly affects the serving performance. Therefore, the UP board is clearly advantaged by its network setup as it is connected in the same LAN of the served user.

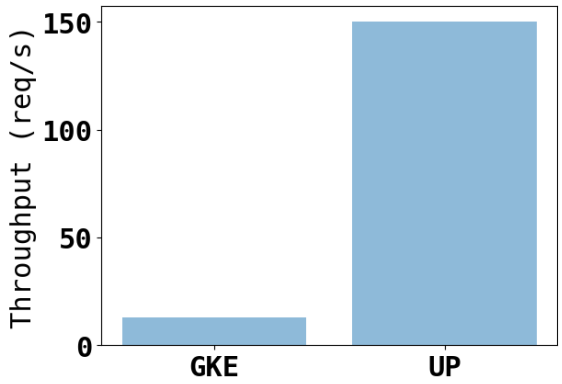

Figure 17: Throughput of the AI model serving with the model “random forest” and request payload from MNIST (one user, one pod)

5 Conclusion and discussion

This chapter will recall the previous work and draw a conclusion based on the results obtained in the previous sections. The conclusion drawn in this chapter is not only a summary of the findings presented before, but also an attempt to answer the research questions raised in the first chapters of the thesis.

First, two AI model serving systems, Tensorflow Serving and Clipper, are investigated. Tensorflow Serving has been in production for relatively long time and its quality and stability are well tested and verified. There are quite good documentations for learners and developers, and it is also widely used and integrated in many machine learning tools. It provides out-of-the-box integration with Tensorflow models, but needs more work to serve with other types of models. In contrast, Clipper can support more types of models conveniently, and it provides the possibility to add custom logic before prediction requests reach model containers. However, considering that this is a newer serving system, its use in a production environment is currently not recommended although it is worth keep following Clipper development.

Next, we describe the deployment methodologies of the AI model serving in a cloud environment and in a single-board computer that could easily be used as edge unit. Due to the fact that we worked on heterogeneous hardware, a unified solution from software perspective – meaning deploying the services as microservices with Docker and k8s – is adopted to abstract the resources on top of the OS and simplify the deployment. For the edge environment we use an Intel UP board and a k8s cluster is started with microk8s. For the cloud setup, a production ready GKE is chosen to facilitate the entire deployment. On top of the k8s cluster, we have similar configurations in edge and cloud. A k8s deployment including one or more pods is started, and the pods download the model images that we made and uploaded in the Docker hub, already trained and ready to execute the AI model serving. A k8s service is created to work as balance the load of the requests to the different pods and expose the model serving outside. From this perspective, there is a difference on the service exposure in the two environments, as the edge setup uses k8s NodePort and exposes its service in the LAN, while the cloud setup uses a k8s LoadBalancer, exposing its service with a public IP on the Internet.

Then, we run several experiments in order to estimate the performance of the serving of several AI models both on edge and cloud environments. Edge computing benefits from lower network latency, while the cloud setup benefits from more powerful hardware and higher computation capacity. However, we found out that the edge environment introduce some advantages in the following situations:

- High-throughput applications (e.g. based on logistic regression), meaning that the time required for processing the serving requests is relatively low, and the computation capacity is not fully saturated.

- When a single user is served. However, a small number of users can benefit

of a performing edge-based serving for very high-throughput applications (e.g. based on logistic regression).

- When prediction caching mechanisms are enabled.
- When the AI model serving requires low resources consumption.
- When high-payload serving requests are issued – like in the case of high resolution images –, as the network latency for handling such payloads from the cloud can dramatically increase.
- When time sensitive AI applications are requested, as the edge setup can ensure lower response time deviation.

On the contrary, cloud outperforms the edge in the following cases:

- Low-throughput applications (e.g. object detection), meaning that the time required for processing the serving requests is relatively long.
- When multiple users are served, as the cloud can increase its resource usage and the influence of network latency is reduced.

In addition, the effect of the allocated k8s pods number is also investigated in the empirical analysis. Adding more k8s pods should be carefully considered, as the performance may deteriorate if the model image is large and the resources for model serving are restricted. However, allocating more k8s pods could be beneficial in the following situations:

- In a cloud environment, as adding more pods allows to use the available resources in the other k8s nodes.
- If an algorithm is designed to use only one core, more pods allow to handle additional instances of the same algorithm and, therefore, to make use of multiple cores.
- If the model serving needs to rely also on external resources and/or dependencies, more pods can ensure higher responsiveness for such external services.
- There is an under-utilization of resources and, at the same time, many requests need to be served.

Future work

One direction to continue the work done in this thesis is to deploy AI model serving tasks on different hardware, e.g. GPU or TPU. Google has released a single-board computer with a removable system-on-module featuring edge TPU [78], which seems very promising to run AI at the edge.

Another interesting aspect to investigate would be to measure the performance of Tensorflow Serving using its gRPC interface. gRPC is based on HTTP/2, which

can send multiple requests in parallel with one single TCP connection. The L4 load balancer used in the cloud setup provided by GKE is not suitable for the usage of a gRPC interface, since it cannot load balance multiple requests coming from the same TCP connection. Differently, a L7 load balancer (ingress) is needed.

Moreover, measurement of the serving with additional models owning completely different features can also be considered as the next step.

References

[1] ETSI, "Mobile Edge Computing - Introductory Technical White Paper", Sep 2014.

[2] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A Survey on Mobile Edge Computing: The Communication Perspective", in *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2322-2358, Aug 2017.

[3] Y. Zhang, J. Ren, J. Liu, C. Xu, H. Guo, and Y. Liu., "A Survey on Emerging Computing Paradigms for Big Data," *Chinese J. Electronics*, vol. 26, no. 1, pp. 1-12, 2017.

[4] J. Ren, H. Guo, C. Xu, and Y. Zhang., "Serving at the Edge: A Scalable IoT Architecture Based on Transparent Computing," *IEEE Network*, 2017.

[5] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for VM-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14-23, Oct.-Dec 2009.

[6] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, "Fog computing: A platform for Internet of Things and analysis," in *Big Data and Internet of Things: A Roadmap for Smart Environments*. Springer, Mar 2014. pp. 169-186.

[7] K. Wang, Y. Wang, Y. Sun, S. Guo, and J. Wu, "Green industrial internet of things architecture: An energy-efficient perspective," *IEEE Communications Magazine*, vol. 54, no. 12, pp. 48-54, 2016.

[8] J. Wu, S. Guo, J. Li, and D. Zeng., "Big data meet green challenges: Greening big data," *IEEE Systems Journal*, vol. 10, no. 3, pp. 873-887, 2016.

[9] J. Wu, S. Guo, J. Li, and D. Zeng., "Big data meet green challenges: Big data toward green applications," *IEEE Systems Journal*, vol. 10, no. 3, pp. 888-900, 2016.

[10] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and communication offloading," *IEEE Communications Surveys & Tutorials*, 2017.

[11] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta and, D. Sabella, "On multi-access edge computing: A Survey of the emerging 5g network edge architecture & orchestration," *IEEE Communications Surveys & Tutorials*, 2017.

[12] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-Efficient Learning of Deep Networks from Decentralized Data," in *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, 2017.

[13] I. Stoica, D. Song, R. A. Popa, D. A. Patterson, M. W. Mahoney, R. H. Katz, A. D. Joseph, M. Jordan, J. M. Hellerstein, J. Gonzalez, K. Goldberg, A. Ghodsi, D. E. Culler, and P. Abbeel, "A berkeley view of systems challenges for ai," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2017-159, Oct 2017.

[14] B. Tang, Z. Chen, G. Hefferman, S. Pei, T. Wei, H. He, and Q. Yang, "Incorporating intelligence in fog computing for big data analysis in smart cities," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 5, pp. 2140-2150, Oct 2017.

[15] H. Li, K. Ota, and M. Dong, "Learning iot in edge: Deep learning for the internet of things with edge computing," *IEEE Network*, vol. 32, no. 1, pp. 96-101, Jan 2018.

[16] X. Chen, Q. Shi, L. Yang, and J. Xu, "Thriftyedge: Resource-efficient edge computing for intelligent iot applications," *IEEE Network*, vol. 32, no. 1, pp. 61-65, Jan 2018.

[17] J. Davidson, B. Liebald, J. Liu, P. Nancy, T. Van Vleet, U. Gargi, S. Gupta, Y. He, M. Lambert, B. Livingston, and D. Sampath, "The YouTube video recommendation system," *RecSys*, pp. 293-296, 2010.

[18] T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich, "Web-Scale Bayesian Click-Through rate Prediction for Sponsored Search Advertising in Microsotf's Bing Search Engine," *ICML*, pp. 13-20, 2010.

[19] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Pillips, E. Davydov, D. Golovin, S. Chikkerur, D. Liu, M. Wattenberg, A. M. Hrafnkelsson, T. Boulos, and J. Kubica, "Ad click prediction: a view from the trenches," in *KDD*, pp. 1222, 2013.

[20] R. Lerallut, D. Gasselin, and N. Le Roux, "Large-Scale Real-Time Product Recommendation at Criteo," in *RecSys*, pp. 232, 2015.

[21] X. Lei, A. W. Senior, A. Gruenstein, and J. Sorensen, "Accurate and compact large vocabulary speech recognition on mobile devices," *INTERSPEECH*, pp. 662-665, 2013.

[22] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. "Tensorflow: Large-scale machine learning on heterogenous systems," 2015.

[23] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the ACM International Conference on Multimedia*, pp. 675-678, ACM, 2014.

[24] S. J. Young, G. Evermann, M. J. F. Gales, T. Hain, D. Kershaw, G. Moore, J. Odell, D. Ollason, D. Povey, V. Valtchev, and P. C. Woodland, *The HTK Book, version 3.4*, Cambridge University Engineering Department, Cambridge, UK, 2006.

[25] J. Langford, L. Li, and A. Strehl, "Vowpal wabbit online learning project," 2007.

[26] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[27] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[28] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A Low-Latency Online Prediction Serving System," *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 613-627, 2017.

[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[30] V. Mnih, N. Heess, A. Graves, et al, "Recurrent models of visual attention," in *NIPS*, pp. 2204-2212, 2014.

[31] AWS cloud computing. https://aws.amazon.com/what-is-cloud-computing/

[32] Redhat public cloud. https://www.redhat.com/en/topics/cloud-computing/what-is-public-cloud

[33] The NIST definition of Cloud Computing. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

[34] GKE. https://cloud.google.com/kubernetes-engine/

[35] CloudFlare edge computing. https://www.cloudflare.com/learning/serverless/glossary/what-is-edge-computing/

[36] S. J. Russell, P. Norvig, "Artificial Intelligence: A Modern Approach, 3rd edition," *Pearson*, 2010, pp. 706.

[37] Scikit-learn. https://scikit-learn.org/stable/

[38] "Speed/accuracy trade-offs for modern convolutional object detectors." Huang J, Rathod V, Sun C, Zhu M, Korattikara A, Fathi A, Fischer I, Wojna Z, Song Y, Guadarrama S, Murphy K, CVPR 2017.

[39] Tensorflow Object Detection API GitHub. https://github.com/tensorflow/models/tree/master/research/object_detection

[40] Tensorflow website. https://www.tensorflow.org/

[41] A. Krizhevsky, I. Sutskever, G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances in Neural Information Processing Systems 25 (NIPS 2012)*, pp. 1097–1105, 2012.

[42] S. Ren, K. He, R. Girshick, and J. Sun, "Fast R-CNN: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, pp. 91-99, 2015.

[43] J. Dai, Y. Li, K. He, and J. Sun, "R-FCN: Object detection via region-based fully convolutional networks," *arXiv preprint arXiv:1605.06409*, 2016.

[44] C. Szegedy, S. Reed, D. Erhan, and D. Anguelov, "Scalable, high-quality object detection," *arXiv preprint arXiv:1412.1441*, 2014.

[45] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," In *European Conference on Computer Vision*, pp. 21–37, Springer, 2016.

[46] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *arXiv preprint arXiv:1506.02640*, 2015.

[47] Logistic regression introduction. https://towardsdatascience.com/introduction-to-logistic-regression-66248243c148

[48] T. K. Ho, "Random decision forests," *Proceedings of 3rd International Conference on Document Analysis and Recognition*, Montreal, Quebec, Canada, vol. 1, pp. 278-282, 1995.

[49] T. K. Ho, "The random subspace method for constructing decision forests," in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 8, pp. 832-844, Aug. 1998.

[50] Tensorflow Serving website. https://www.tensorflow.org/tfx/guide/serving

[51] Clipper website. http://clipper.ai/

[52] Tensorflow Serving in Kubeflow. https://www.kubeflow.org/docs/components/tfserving_new/

[53] YAML Wikipedia. https://en.wikipedia.org/wiki/YAML

[54] Docker website. https://www.docker.com/

[55] Virtual machine Wikipedia. https://en.wikipedia.org/wiki/Virtual_machine

[56] V. Singh and S. K. Peddoju, "Container-based microservice architecture for cloud applications," 2017 *International Conference on Computing, Communication and Automation (ICCCA)*, Greater Noida, 2017, pp. 847-852.

[57] D. Namiot and M. Sneps-Sneppe, "On Micro-services Architecture," *International Journal of Open Information Technologies*, 2014, 2(9): 24-27.

[58] Z. Xiao, I. Wijegunaratne and X. Qiang, "Reflections on SOA and Microservices," 2016 4th *International Conference on Enterprise Systems (ES)*, Melbourne, VIC, 2016, pp. 60-67.

[59] DockerHub website. https://hub.docker.com/

[60] Kubernetes website. https://kubernetes.io/

[61] Minikube GitHub. https://github.com/kubernetes/minikube

[62] Mircok8s website. https://microk8s.io/

[63] Google Cloud. https://cloud.google.com/

[64] Docker installation. https://docs.docker.com/install/

[65] COCO website. http://cocodataset.org/#home

[66] Tensorflow detection model download link. https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md

[67] Y. LeCun, C. Cortes, and C. J. Burges, "MNIST handwritten digit database," 1998.

[68] OpenML website. https://www.openml.org

[69] AVX wikipedia. https://en.wikipedia.org/wiki/Advanced_Vector_Extensions

[70] gRPC. https://grpc.io/

[71] Tensorflow Serving Dockerfile. https://github.com/tensorflow/serving/blob/master/tensorflow_serving/tools/docker/Dockerfile

[72] Tensorflow Serving Installation. https://www.tensorflow.org/tfx/serving/setup

[73] Anaconda. https://www.anaconda.com/

[74] Anaconda installation. https://docs.anaconda.com/anaconda/install/

[75] Clipper bug. https://github.com/ucbrise/clipper/issues/695

[76] CIFAR. https://www.cs.toronto.edu/~kriz/cifar.html

[77] Heapster GitHub. https://github.com/kubernetes-retired/heapster

[78] Coral Dev Board. https://coral.withgoogle.com/

A Python script example to measure the performance of AI model serving

```
import numpy as np
import queue
import threading
import time
import os
import json
import statistics
import requests


def parse_data_array(file):
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    data = dict[b'data']
    return np.rollaxis(data.reshape(10000, 3, 32, 32), 1, 4)


class TestPerformance:

    def __init__(self, server_address, n, number_threads):
        self.server_address = server_address
        self.n = n
        self.number_threads = number_threads
        self.data = parse_data_array('data_batch_1')
        self.q = queue.Queue(maxsize=self.n)
        self.out_dict = {}
        self.total_time = 0

    def prepare(self):
        for index in range(self.n):
            self.q.put(index)

    def consume(self):
        while True:
            try:
                idx = self.q.get_nowait()
            except queue.Empty:
                break
            else:
                #print(idx)
                image_data = self.data[idx, :, :, :]
                shape = image_data.shape
                height = shape[0]
                width = shape[1]
                color = shape[2]
                image_array = image_data.reshape((1, height, width,
                    color))
                start_time = time.time()
```

```
49                  response = requests.post(
50                      "http://%s/v1/models/%s:predict" %
51                      (self.server_address, '
                            universal_ssd_inception_v2_coco'),
52                      data=json.dumps({'inputs': image_array.tolist()})
                            )
53                  # result = response.json()
54                  # print(response.status_code)
55                  # print(result)
56                  total_time = time.time() - start_time
57                  self.out_dict[idx] = total_time
58                  self.q.task_done()
59
60      def start_till_finish(self):
61          start_time = time.time()
62          for i in range(self.number_threads):
63              t = threading.Thread(name="Thread-" + str(i),
64                                   target=self.consume, args=())
65              t.start()
66          self.q.join()
67          self.total_time = time.time() - start_time
68
69      def mean_latency(self):
70          latencies = [self.out_dict[key] for key in self.out_dict]
71          return statistics.mean(latencies)
72
73      def standard_deviation(self):
74          latencies = [self.out_dict[key] for key in self.out_dict]
75          return statistics.stdev(latencies)
76
77      def throughput(self):
78          return self.n/self.total_time
79
80      def test_time(self):
81          return self.total_time
82
83      def number_of_samples(self):
84          return self.n
85
86  # cloud
87  server = '35.228.29.230:80'
88  # edge
89  #server = '10.0.0.222:30851'
90
91  n_samples = 5000
92  n_threads = 2
93
94  test = TestPerformance(server, n_samples, n_threads)
95  test.prepare()
96  print("Preparation is done.")
97  test.start_till_finish()
```

```
98
99
100 print("{} images".format(test.number_of_samples()))
101 print("{} seconds past".format(test.test_time()))
102 print("Throughput is {} req/s".format(test.throughput()))
103 print("The mean response time is {} seconds".format(test.mean_latency
    ()))
104 print("The stdev of response time is {} seconds".format(test.
    standard_deviation()))
```